Elicitation of Formal Software Specifications from an Object-Oriented Domain Model

THESIS

Timothy Karagias
Captain, USAF

AFIT/GCS/ENG/96D-14

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/96D-14

Elicitation of Formal Software Specifications from an Object-Oriented Domain Model

THESIS

Timothy Karagias
Captain, USAF

AFIT/GCS/ENG/96D-14

AFIT/GCS/ENG/96D-14

Elicitation of Formal Software Specifications from an Object-Oriented Domain Model

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Timothy Karagias, B.S.Computer Science

Captain, USAF

December 17, 1996

Acknowledgements

First and foremost, I would like to thank my wife, Susan, for supporting me in my career choices. I would also like to thank my children: Tim Jr., Steven, Mark, and Sarah for helping me keep my perspective on life. Finally, I would like to thank my thesis advisor, Dr. Hartrum, for pulling me through this challenging endeavor.

Timothy Karagias

Table of Contents

List of Figures

AFIT/GCS/ENG/96D-14

Abstract

The ability to provide automated support for the generation of formal software specifications would lead to decreased software development time. By *eliciting* the needed information from a software developer and *harvesting* the proper parts of a domain model, a software specifications document could be created. This research establishes the feasibility of producing customized software specifications based on an object-oriented domain model. The research was conducted in three phases. The first phase was to define the requirements for the Elicitor Harvester. Those requirements were balanced between the capabilities of the existing Knowledge Based Software Engineering (KBSE) software used at AFIT and the needs of the Elicitor Harvester system. The second phase consisted of creating a design capable of meeting those requirements. The design was open enough to use the existing software and flexible enough to evolve in an incremental manner. The final phase involved the implementation and testing of a feasibility demonstration of the Elicitor Harvester system. Specifications were successfully generated from two significantly different domain models.

Elicitation of Formal Software Specifications from an Object-Oriented Domain Model

I. Introduction

1.1 Background

The software industry is debating the use of formal methods in the production of software. This method is often tedious due to the manual nature of the process. Software houses tend to avoid the process due to its perceived difficulty and lengthiness. However, many studies have shown how the cost effectiveness of reducing defects in the earlier stages of the software life-cycle, by formal methods or quality control measures (9), are well worth the time and effort. Among the gains are reduced development time and maintenance costs.

Another area of interest is the reuse of software components at every phase of the software process. In particular, the cost benefits of reuse are most prevalent at the highest levels with architecture reuse being the current goal (2). To help aid in this goal, domain modeling is used to help identify reusable components in specific problem domains. Domain modeling involves the identification of components common across a domain. Domain experts (Figure 1.1) develop a model in some representation language. They capture the essence of the domain, including how the pieces interact with each other. This model gives a generic architecture of the domain. The important thing to note is that the domain expert is not necessarily a software person but truly knowledgeable in the specific domain.

After the domain model is captured, software developers interested in building a system in that domain can use the model in developing the formal specifications for that specific system. This customization of the domain model (Figure 1.1) leads to formal specifications for the problem at hand. This step can be thought of as the *elicitor* step. The domain model is used to elicit further information pertinent to the problem at hand. The software developer fills in the holes present in the domain model and produces the formal specifications needed for the system under development.

From the formal specification, a design for the software system can be developed. Information from the specification is used to develop the components needed to build the specific problem (Figure 1.1). From the design, the system will be implemented in an executable language.

In the spirit of reuse, software developers often *harvest* software components which can be used to solve the problem at hand. This harvesting can take place at any step in the software development process. For example, a developer may look for a domain model he or she has done in the past that can be applied to the new system. That old system may also contain the formal specifications which can be used with little or no modification. After the specifications are complete, the software developer may find reusable software designs that can be used, along with the code necessary to implement the design. The bottom line is that reusable components can be harvested at any point in the development of a new system.

As with most phases in the software process, automated tools can aid immensely in the production of formal specifications and identifying component reuse. The Knowledge Based Software Engineering (KBSE) research group at the Air Force Institute of Technology (AFIT) continues to conduct research into all phases of domain oriented tool development. Work has already been done in the area of building formal domain models specified in Z schemas using LaTeX syntax that are parsed it into a REFINE abstract syntax tree (11) . This work has been duplicated using algebraic languages (3). The ability to manipulate a REFINE abstract syntax tree to generate a specification from a domain model has also been demonstrated with AFIT's Architect system (13). This indicates that the ability to go from domain analysis to a software specification exists. However, many gaps in the system are still prevalent.

1.2 Problem

The ability to generate formal specifications for a specific problem from a domain model is still a manually intensive process. Because of this, many details may be missed or overlooked. As the software process progresses with those missed details, so does the cost of fixing them. By automating the process, the details can be checked to ensure that they are carried along in the process.

Lack of automation across the process leads to loss of information as well as dramatic shifts in the representation of the process. Automation allows for a consistent flow of information and representation across the software development process. Often times, changing from one software development phase to the next, calls for a switch in the representation of the information. For example, a high level design may lead to the production of a written design document while the detailed design may take the form of an automated hierarchical diagram. This leads to loss of possibly crucial information because the new representation

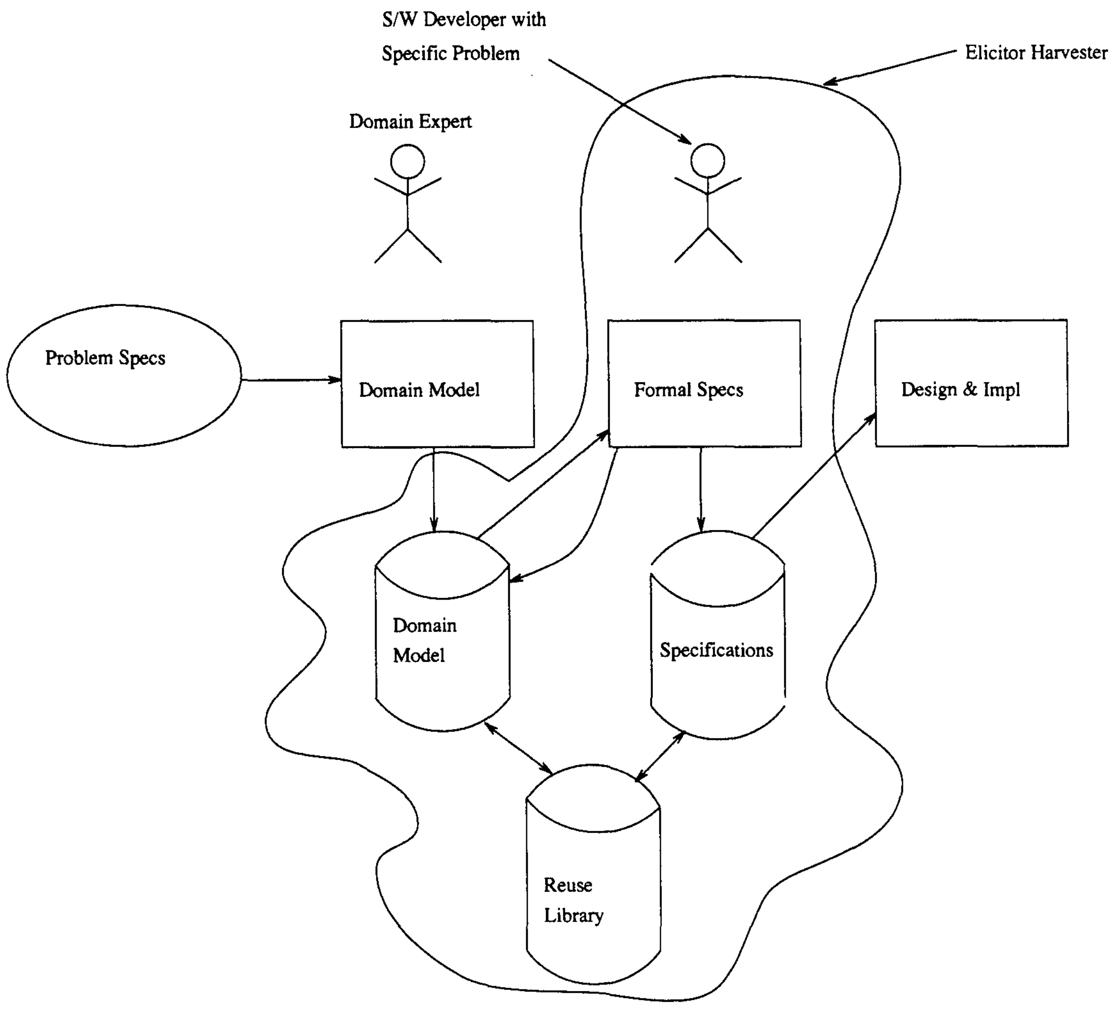

Figure 1.1 Domain Model based Software Development

may not have any way to reflect a piece of vital information. Changes between manual and automated processes must be avoided as much as possible.

The process of producing a domain model and formal specifications from that domain model is not yet a well formed process. Because of this, there is often a lack of consistency. Consistency in the implementation is an important goal when it comes to building software components. Domain specifications must be consistent to allow for the matching of current information to existing information in order to develop or reuse the proper formal specifications. Formal specifications must be consistent to allow for matching of reusable specifications as well as reusable design components.

Identification of software components is also a big problem. Without a way to file reusable components consistently and correctly, finding those components when needed is difficult and time consuming. Information must be classified in a consistent manner and must reflect the level of a component, as well as the behavior of the component in order for it to be reused.

Problem Statement: *Demonstrate the ability of KBSE technology to accurately elicit the requirements from a user and combine/manipulate objects in the domain model to generate a formal specification for the specific problem at hand.*

1.3 Initial Assessment of Past Effort

AFIT's KBSE research has produced many of the elements needed in automating the software process using formal methods. Domain models can be parsed from Z schemas into REFINE Abstract Syntax Trees (AST) as the foundation. Specifications are entered into a LaTeX file using Z schemas. This is a difficult process which calls for a detailed knowledge of Z and LaTeX. A similar tool exists for the use of Larch specifications in place of Z schemas (3). However, no steps have been taken to eliminate the difficulties in interfacing with the system. From this point, the abstract syntax trees from Larch can be transformed into an AST for the algebraic language O-SLANG[7]. This AST can then be transformed to Slang which allows for further design refinement (3). At this point many tools are available for theorem proving and analysis.

Identification of reusable objects via requirements elicitation shows that requirements can be obtained interactively[8]. Although many of the principles may be useful in the area of user interface and possibly in questioning techniques, this work was focused on a narrow domain. Producing specifications from an AST using a domain model has been demonstrated with AFIT's Architect system. This system used a domain specific language for construction of an AST and produced specifications in a restricted format. To summarize, the construction of AST from Z or Larch specifications appears to apply in any domain. On the other hand, construction of specifications based on domain models has, up to this point, focused on very narrow domains. No attempt has been made to integrate the two processes to generate formal specifications.

1.4 Assumptions

At this time, the KBSE software involved with the production of the abstract syntax tree from Z and Larch specifications was assumed to be fairly complete. It was also assumed that all software was well documented. The Architect system documentation was also assumed to have in-depth information on the process of converting abstract syntax trees into system specifications. Rumbaugh's OMT must also be sufficient to model all aspects of any covered domain. If OMT is not sufficient, extensions must be incorporated.

1.5 Scope

The modeling of all possible problem domains would be too grand of an effort at this point. The goal was to show that domain requirements could be elicited in a thorough manner and that the system requirements could be produced from the existing domain model. By limiting the domain representation to object oriented Z specifications, such as the existing school and missile Z specifications, it was possible to concentrate on solving the Elicitor Harvester problem and not how to represent a domain model. However, the system was designed with the concept of interchangeable representations, as long as those representations could be parsed into the same abstract syntax trees.

1.6 Research Objective

The overall objective of this research was to demonstrate the ability to partially automate the process of generating formal specifications from a domain model using a knowledge based assistant capable of eliciting all necessary information from the user to produce valid system specifications.

1.7 Approach

To meet the proposed research objective the following approach was followed:

1. *Gain an understanding of Knowledge Based Software Engineering* - Conduct a survey of current software engineering literature to gain insight into what KBSE entails, to

include domain analysis and modeling and formal methods. This step was necessary to indicate what was or was not needed in the task at hand

2. *Study current issues in the area of requirements gathering* - Reach a deep understanding in the process of interviewing and gathering requirements. This was necessary for the design of the elicitor interface. Proper questioning and identification of requirements gaps was essential in the elicitation of information from the user. This allowed for the development of an algorithm to be used in the elicitation of requirements necessary for formal specifications of domain modeling.

3. *Analyze current state of KBSE software* - Perform an analysis of existing software pertinent to the problem. All capabilities currently available were identified in order to gather proper requirements to aid in the identification of limitations as well as the pieces needed for the implementation of the elicitor harvester.

4. *Define Elicitor Harvester system requirements* - Produce the system requirements for the Elicitor Harvester after discovering what was available and analyzing what else was needed. This step included what could or could not be done with the Elicitor Harvester and the existing KBSE research software.

5. *Perform domain modeling* - Analyze the present state of a given problem domain to include past domain models and present work. This was necessary to demonstrate the actual execution of the Elicitor Harvester on a specific domain. Without focusing on a specific domain, the project would have been too large to perform in the allotted time.

6. *Develop Elicitor Harvester* - Implement a small scale system version of the Elicitor Harvester. It was necessary to demonstrate the feasibility of the requirements gathering and production via an automated tool. Specific problem areas were identified as well as the possibility of expanding the number of domains the Elicitor Harvester can handle.

7. *Test Elicitor Harvester* - Run a small scale test using existing specifications to uncover successes and problem areas with the Elicitor Harvester design or implementation.

8. *Show feasibility of expanding domains* - Show how the Elicitor Harvester can be expanded over several domains and possible future direction of the system. This aided in identifying the shortcomings as well as promising areas of the Elicitor Harvester software.

This approach allowed for the successful construction of the necessary foundation for a successful research project, covered the pertinent areas, and fulfilled the end goal of the research.

1.8 Conclusion

This introduction section explored the need of automated tools in the software arena. It also covered the advances made at AFIT in the world of Knowledge Based Software Engineering. The assumptions, scope, objectives and approach was also laid out for the Elicitor Harvester research effort. Chapter 2 discusses the current work in domain modeling. This includes concepts and current systems that take advantage of high level reuse. Chapter 3 lays out the requirements for the Elicitor Harvester. Chapter 4 contains the approach and design of the Elicitor Harvester system. Chapter 5 looks at the implementation and testing. This includes support software and testing strategy. Finally, Chapter 6 discusses the results and conclusions of this research, as well as future recommendations.

II. Literature Search

2.1 Introduction

This chapter examines research and information relevant to the design of the Elicitor Harvester described in Chapter 1. Broad coverage of topics in software engineering, computer science, and artificial intelligence was required. Based on the objectives of the research, three different types of articles were considered: domain analysis from an overall implementation point of view, formal methods used in domain modeling, and artificial intelligence based systems.

2.2 Domain Analysis Overall Implementation

The term "Domain Analysis" was coined during the DRACO system development effort under a grant from the National Science Foundation and the Air Force Office of Scientific Research(AFOSR) [Neigh 89]. The intent of domain analysis is to identify and classify commonalities within specific domains. While the goal of the Elicitor Harvester was the production of system specifications and the articles in this section focus on reuse of objects and code, much of the information discussed in these articles can be adapted to formal specification reuse. The following paragraphs in this section cover domain analysis from an overall implementation point of view.

2.2.1 The First Step: Planning. This article looks at domain analysis from the context of software reuse, with specific emphasis on Ada and DOD systems. It gives lessons learned based on experience gathered while implementing reuse for the Program Executive

Office for Standard Army Management Information Systems (PEO STAMIS). The most important point it emphasizes is the need to incrementally implement the plan (6).

Having an approach is the first step, according to Owens. For the PEO STAMIS project, eight key aspects of their approach were identified. They are:

1. Identification and description of domain (and sub-domain) boundaries.
2. Initiation and support of one (or more) short-term Reuse Pilot Project(s).
3. Development of generic architectures and domain-specific models.
4. Identification of generic and domain-specific high-demand reuse categories.
5. Identification of horizontal and vertical reuse opportunities within the domain.
6. Implementation and population of a Domain Knowledge Database.
7. Quantification of potential short-term and long-term cost avoidance.
8. Preparation of a multi-year, domain-specific Reuse Implementation Plan.

In addition to the above general approach guidelines, five steps were developed for the Domain-Specific Software Architectures (DSSAs) domain engineering process. They are:

1. Define the scope of the domain
2. Define (or refine) domain-specific concepts and requirements
3. Define (or refine) domain-specific design and implementation constraints.
4. Develop domain architectures and models.
5. Produce or gather reusable products.

The key to the domain-specific reuse plan (Figure 2.1) according to Owens is to incorporate domain engineering activities to include domain analysis, domain design, and domain implementation through the use of reuse libraries and other domain environment support resources. It was also mentioned that three specific documents should be created in support of the environment. The first document is the Domain Definition Report (DDR). Its purpose is to identify the boundaries of the domain, to define principal sub-domains, and to determine relationships and links between the sub-domains. Other documents include a Reuse Opportunities Report (ROR) which identifies reuse opportunities within a specific domain, and a reuse implementation plan (RIP) which presents a strategy to implement reuse within a specific domain.

2.2.2 Developing Software Within A Domain. This article focuses on the activities of the domain analyst and speaks mostly of code reuse. Limitations are identified in the existing methods of domain analysis along with an approach to overcome those problems in the Domain and Object Oriented Reuse (DOOR) system (1). The three limitations of previous efforts include lack of informal techniques using ad hoc approaches, no tool support and inconsistent terminology, and the separation of domain analysis from system analysis. The authors propose the merging of the two stages and using object oriented techniques to structure the application domain into sub-domains to facilitate the identification and description of all functions in a specific domain. As in the previous article, the importance of identifying boundaries was mentioned as a key ingredient to success.

The domain analyst's job was divided into domain assessment and domain modeling. The first part addresses the preparation of the domain model resources and the building

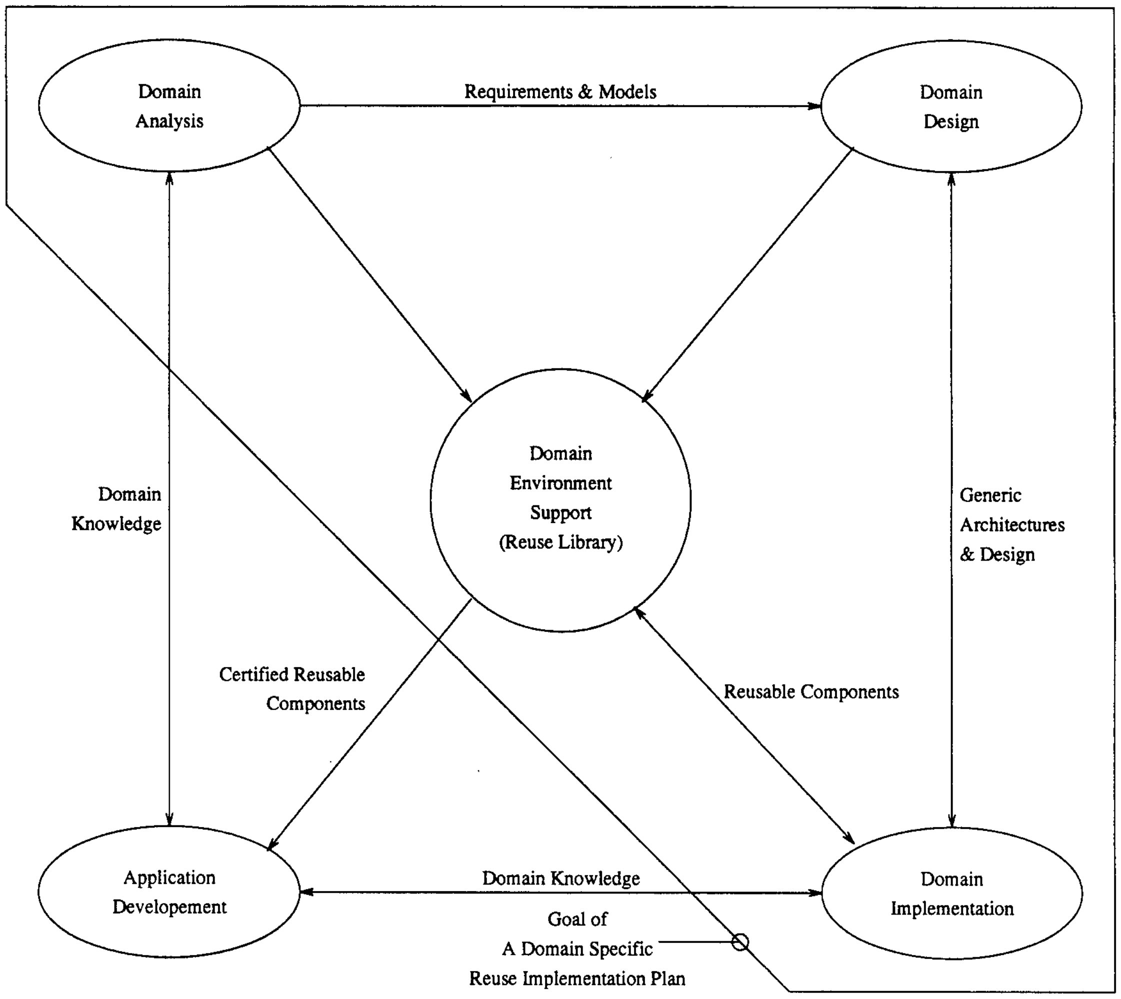
Domain Analysis
Requirements & Models
Domain Design
Domain Environment Support (Reuse Library)
Domain Knowledge
Generic Architectures & Design
Certified Reusable Components
Reusable Components
Application Developement
Domain Knowledge
Domain Implementation
Goal of A Domain Specific Reuse Implementation Plan

Figure 2.1 A Reuse Plan Can Incorporate Domain Engineering Activities [Owens93]

of a framework for modeling the components. The second part addresses the actual construction of a specific domain model. First, domains can be classified as abstraction levels according to common features/objects in each level. The common objects are separated from the rest of the objects and are referred to as the Domain Kernel (DK). From there, the rest of the objects are classified in a step by step manner into sub-domains until no further decomposition is needed. This is analogous to a battle field simulation system where the simulation and the players are used as the kernel with sub-classifications of players such as tanks and missiles.

2.2.3 The Avionics Domain Application Generation Environment (ADAGE).

This paper focuses on domain modeling in the avionics domain and the work was sponsored by the Department of Defense Advanced Research Projects agency along with the U. S. Air Force Wright Laboratory Avionics Directorate (2). The premise of ADAGE is that the problems facing the avionics domain, such as guidance and navigation, are well understood. Because of this fact many new systems can be built by combining and adapting existing components to meet the new requirements. Of important note in this article is the explanation and combination of the two types of domain models, referred to as integrative and generative.

Integrative models involve the automatic production of specifications for a specific target system. Integrative models consist of an integrated set of sub models which provide alternate views to include generalization/specialization, state transition diagrams, aggregation hierarchies, and features. However, they do not provide the ability to include or exclude optional objects, relationships, and states when specifying a target system. The feature sub-model is used to capture the rule base (knowledge base) that constrains the selection to only reasonable combinations. In other words, only selections that make sense are allowed.

Generative models relate more with software architecture problems such as identifying fundamental abstractions of a domain, creating libraries of plug compatible and inter-operable software building blocks for a domain (ADAGE calls them building blocks), defining the knowledge representation (ADAGE calls them type equations) and classifying the different communication protocols (ADAGE calls them realms) that combine the different building blocks. The ADAGE generative model contains over 40 different realms and over 350 components. The largest problem faced by ADAGE is that many times a syntactically correct type equation may not make any semantic sense.

To help alleviate this problem, ADAGE is looking to a tool called Variational Attribute Grammars (VAG) to move towards the integrative model. VAG is used much in the way Z is used in system specification. However VAG is a functional programming language used as part of partial designs. The intent is to declare yet-to-be-built components as variables for future development. Its weak point is that human designers are still depended on to make major decisions.

2.2.4 Model-Based Reuse Repositories. This paper discusses results and lessons from the development of the Comprehensive Approach to Reusable Defense Software (CARDS) Program (7) which was sponsored by the U. S. Air Force and performed by Unisys Corporation. The difference between this paper and others in this section is that it aims not at component reuse but at the model-based repository approach. By using this approach, the decision of how to fit repository components into a new system is moved from the shoulders of the software engineer onto the actual system. The burden of how different components interface with the rest of the system and how to change those components is assumed by the reuse system. The goal of CARDS is to move up the level of reuse from the modules/objects to the requirements and design level. As in the earlier articles, the use of a generic architecture, specifically the DSSA, plays a major role in the design of the system.

The Domain Specific Software Architecture (DSSA) is a way of capturing information within any one specific domain model. At a minimum it needs to identify component classes, connections, constraints, and rationale.

Component classes come from the partitioning of the overall system functionality which is captured in the domain model. This partitioning is influenced by a set of available products that can meet the needed functionality. For example, components within a drive-train can be used for tanks and for cars. Both drive-trains have similar functionality and may even be interchangeable.

Connections capture how component classes are connected. This could capture data flow, direction, and type. However, care must be given to ensure interface configurations are flexible. The goal is to alter the connections based on various constraints involved with the matching of two different component classes. For example, the drive-train may have different parameters to pass when matching up two different engines with two different transmissions.

Constraints give flexibility to the system. By allowing for qualification of selection criteria, systems can change based on previous choices. The goal is to specify alternative and optional constraints in the component classes. An example of this would be the building of a propulsion system. If a rocket engine is used instead of a jet engine, there is no purpose in selecting a fuel tank or having a refueling function.

Rationales are used to give additional information when selecting among reusable components. From the previous example, suppose information is stored to let the user know the pros and cons of selecting a jet engine over a rocket engine. As the user is preparing to make the decision, this information becomes available to facilitate the choice of propulsion systems.

The final point drawn from the article is in the area of encoding the knowledge base. A key feature to look at is the ability of the language to support the specification of fixed, alternative, and optional attributes for entities represented in the knowledge base. It should also allow for the capturing of dependencies, constraints, and other relations.

2.3 Formal Representations of Domains

In this section, the focus is on the different types of languages used to capture domains and to develop systems from the existing information. The key issue is capturing the knowledge necessary in constructing the needed system, either in specification format or in an end system. The first paper looks at the use of languages to produce an executable system from the domain model. The second paper discusses several different languages used to describe design components in a reuse system. The last paper of this section discusses the use of algebraic specifications to capture knowledge representation.

2.3.1 Language-Oriented Programming. The idea behind language-oriented programming (12) is that large projects should develop their own formal language to suit their specific domain. It recommends a three-tiered approach. First, develop a very high level formal language that captures the uniqueness of the domain. Second, implement the system using a middle level language. Third, implement a compiler/interpreter for the language. The claim is that this approach takes advantage of domain analysis, rapid prototyping, maintenance, portability, and reuse of development work while also providing high development productivity. This type of approach is very costly. However, it does show the importance of developing hierarchies in the construction of any reuse system.

2.3.2 Component Description Languages and Models. This paper discusses the key aspects to examine when looking for models and languages to implement a reuse/domain analysis system. Two different areas of concern were addressed, component models and component description languages (14), as well as making sure that the two compliment each other. The following paragraphs discuss these two areas in more detail.

Component models are abstract descriptions for components in specific domains. Two different component models are discussed and compared/contrasted. They are the 3C model and the REBOOT model. The 3C model is based on work by W. Tracz and looks at concepts, content, and context. The concept gives the abstract description of what a component does and should cover pre and post conditions. Content gives more detail of a component such as how the functions are actually implemented. Context is further divided into three groups. The first is the conceptual context which covers relationship issues to include interface and semantics. The second is operational context which gives data characteristics similar to ADTs. Finally, implementational context gives a description of how components depend on other components. REBOOT, on the other hand, looks at *dependencies*, *abstractions*, *operations*, and *operates on*, which are called facets. The *dependencies* facet deals with the relationship of different components much like the context part of the 3C model. The *abstractions* facet describes the object that a component implements. The *operations* facet deals with operations that can be performed on components, such as GetFuelTankLevel on a fuel tank object. Finally, the *operates on* facet deals with parameters in the system and how they interact within a given environment. Selection of a component model should be based on which matches best with the domain being modeled.

Component description languages are used to capture necessary attributes of the components in a specific domain. Their goal is to provide structure to the syntax and semantics of components, much like programming languages do. Some of the better known component description languages include LIL, Act Two, LILEANNA, and Resolve. When selecting a language, some of the features to look for include:

1. How does the language compare with the implementation language? This issue contributes to ease of implementation and trace-ability.

2. How straight forward is the structure and syntax? Some languages are complicated and need to be looked at from several views to be understood.

3. How flexible is the design? Issues such as generic parameterization and the ability to put off future decisions plays a major role in selecting a language to model the domain.

4. How well does it match up with the component model? The easier it is to go from the component model to the component description language, the easier it is to automate.

This article provides guidelines to aid in the selection and justification of the use of languages and models. One factor not discussed though is corporate knowledge. That is, if there is existing infrastructure to include module descriptions, training programs, and widely accepted languages that can be adopted to fit the needs of the project, those languages/models should probably be looked at as a good viable option.

2.3.3 Algebraic Specifications for Knowledge Representation. The use of algebraic specifications as they relate to knowledge representation was examined in this article (4). The weakness of algebraic specifications is that properties are generally handled through comments, hence the meaning is often lost. As a way of capturing those missing properties, the writer came up with a unified algebraic model based on first order logic that incorporates methods to capture those properties. Their language, which is similar to many algebraic languages, is called Formal-E and incorporates extensions to be used in their knowledge base (MANTRA is covered in the next section). These extensions allow decision points to be identified and specifications to be altered dynamically. The end system discussed in this paper has executable systems, via automated tools, as a goal.

2.4 Artificial Intelligence Based Systems

This final section discusses domain analysis and reuse from an artificial intelligence point of view. This is where the largest benefit can be achieved because it allows for systems that can know what is needed and switch component specifications based on other information. The first article discussed is continued from the Algebraic Specifications for Knowledge Representation section and continues with the knowledge representation used in the author's system [Calme94]. The second article discusses the application of uncertainty management to the domain analysis methods. The final article gives an approach to reuse that involves the use of artificial intelligence.

2.4.1 The Knowledge Part of the System. Part of Calmet's system uses an artificial intelligence language known as MANTRA, which is a knowledge representation language. The reasons for picking MANTRA were its ease of alteration and its multi-layered architecture. The first layer is the epistemological level, which gives the formalism necessary to represent facts in the memory of the computer. The second layer is the logical level, which involves the management of the knowledge base to include primitives to store and draw conclusions from those facts stored in the computer. The third layer is the heuristic level, which manages the search space. The key point gathered from this article is that existing systems can be adapted effectively to meet the desire of automating system development.

2.4.2 Uncertainty Management and Domain Analysis. This article focuses on capturing the uncertainty involved with reusable products and using it to aid in domain analysis and design (10). This is done by interpreting this uncertainty as an organizing principle when examining different approaches to reuse. The following are three questions that give an idea of how to handle uncertainty in reuse systems:

1. How does a given work-product really behave in its original context? If work-products are drawn from a single-system context, they may be under- or incorrectly-specified. This may result from requirements being handled in one of the external modules or poor testing.

2. How will a reused work-product behave in a new context? Work products may behave in erratic ways when used in a new context. This results from contextual assumptions made during original system development. The context resulting from the synergy of multiple contexts/constraints may have adverse effects on the work-product when placed in a new environment.

3. How will adaptations affect behavior of work-products? Since the work-products are adapted without a complete understanding of the entire work-product, the dependability of that work-product may be in question.

These uncertainties result in "architecture erosion" in system maintenance and re-engineering. To avoid this problem, proper scoping of domains and components is called for. By scoping the components, along with designing those components for reuse, areas of uncertainty become more clear and resulting systems are more robust. The bottom line is that when designing a reuse system, ensure that the context is considered.

2.4.3 Another Artificial Intelligence Approach to Reuse. This article focuses on the use of artificial intelligence systems in the selection and assembly of reusable components (8). Four phases of software reuse systems were examined and implemented. The first is representation, which addresses the features and attributes needed to model a software component. The second is retrieval, which addresses the selection of modules to be used based on cost effectiveness. The third is adaption, which calls for methods needed to change or alter the functionality or meaning of components. The fourth and final phase is incorporation, which involves the integration of the new and retrieved components. The idea is to find a system that can perform these four steps for less that it would cost to construct those components from scratch. Artificial intelligence is used to judge cost effectiveness of choices. The idea is to compute the costs, on the fly, of assembling and altering modules and comparing them to the cost of starting from scratch. Algorithms to compute cost and to examine other metrics are given. However, the main point drawn from this article is not the AI formalism of the article but the phases and evaluation criteria used in selection of modules.

2.5 Conclusion

These articles established the major ingredients necessary to build an Elicitor Harvester. The first ingredient was the need to scope the domain to a manageable level. CARDS uses this scoping idea to allow for only the necessary information to be present at a given level. The second ingredient was the structuring of the Elicitor Harvester into layers to facilitate incremental construction of the final system. The DOOR system is an excellent example of layering information in a usable way. Language oriented programming also shows the effectiveness of arranging hierarchies when designing reuse systems. The third ingredient was the use of extensions to existing, well accepted systems to facilitate flexibility in the Elicitor Harvester. Guidelines provided from the section on component description languages helped in the selection and justification of the use of languages and models within the Elicitor Harvester. The final ingredient was the ability to capture alternate and optional constraints. DSSA identifies the importance of capturing these alternatives and options. The key to a successful Elicitor Harvester was to provide the important ingredients from all of these systems.

III. Requirements

3.1 Introduction

Chapter 1 introduced the concept of reuse and its importance to the efficient development of software at reduced costs and time. It also discussed the different levels of reuse, such as code, design and specifications, and the need to move to a higher abstraction level. Domain modeling is one way of capturing high level abstraction for reuse. Chapter 1 also covered some of the work going on at AFIT in the area of Knowledge Based Software Engineering (KBSE) and how it contributes to reuse of formal specifications. KBSE also allows for the eventual synthesis of specifications into actual code. The goal of this research effort is to demonstrate the feasibility of using domain modeling and existing KBSE software to derive specifications for specific problems. The following sections of this chapter form the problem, lay out an approach, and discuss the expected results.

To accomplish the development of the Elicitor Harvester, five steps were laid out for the research effort into its feasibility. Further refinements to the approach mentioned in Chapter 1 were needed at this time and they are:

1. Develop the Knowledge Base - This involves altering the existing AFIT domain tree to suit the purposes of the Elicitor Harvester and is discussed in Section 3.2.
2. Identify the Requirements - This step requires identification of all changeable items and gathering all information needed to change that item. This is also discussed in Section 3.3 of this chapter.

3. Design the Elicitor Harvester - This step includes all levels of design, from the overall system down to the low level functions required to change the actual domain specifications. This is discussed in detail in Chapter 4.

4. Coding the Functions - Refine was used to implement all identified functions. This step involved coding the design modules and is discussed in Chapter 5.

5. Testing the System - Test data in the form of domain models were required to exercise the Elicitor Harvester. These domains were altered following a test plan discussed in Chapter 5. Results are discussed in the final chapter.

3.2 Elicitor Harvester Operational Environment

The real effort of this research is to lay the groundwork for the Elicitor Harvester. To do this we need to start with the most basic of requirements, what is the expected input and what is the expected output. Background information is also needed in the area of Object Oriented analysis and the AST used to capture that analysis. This is necessary to understand the derived requirements discussed in the next section.

3.2.1 Input of the Elicitor Harvester. As discussed in Chapter 2, many ways of structuring software have been used. ADAGE uses a building block approach which allows for software products to be used in conjunction with each other if their interfaces match. Other systems, such as DOOR, focus mainly on code reuse and identify reuse items via a data base. At AFIT, the KBSE group uses Abstract Syntax Trees (ASTs) to capture information about the domain and specific objects within the domain. The

Elicitor Harvester adopted the AST to capture the structure of an object-oriented domain model.

The most important part of an Object Oriented Analysis (OOA) in terms of domain modeling is the relationships between objects. These relationships capture the interaction of objects within the domain. The two most important relationships are the aggregate and the association. An aggregate is a set of component objects that make up an aggregate object such as the fueltank, throttle, and jet engine making up a propulsion system. This relationship may capture such information as allowable combinations of the three components. For example, the propulsion system may specify a maximum weight which limits the combination of the three components to meet that limit. An association captures a relationship between two different components such as a relationship between a class offering and a student within the school domain. This relationship may capture cardinality which reflects the fact that many students may be enrolled in a class or that one student may be enrolled in many classes.

The input for the Elicitor Harvester is an AST constructed from an OOA of a domain. Figure 3.1 shows the basic AST used by the domtree and GOMT software at AFIT.

ASTs allow for flexibility in the building of the Elicitor Harvester. Manipulation of items in the tree occur when needed without any adverse effects on the rest of the tree. By using the AST from the GOMT system, compatibility with other AFIT projects was increased. Finally, the feasibility of parsing formal specifications into the AST has been demonstrated. The AST also facilitated an evolutionary approach to the Elicitor Harvester. Figure 3.2 reflects the AST used to represent the generic domain model used

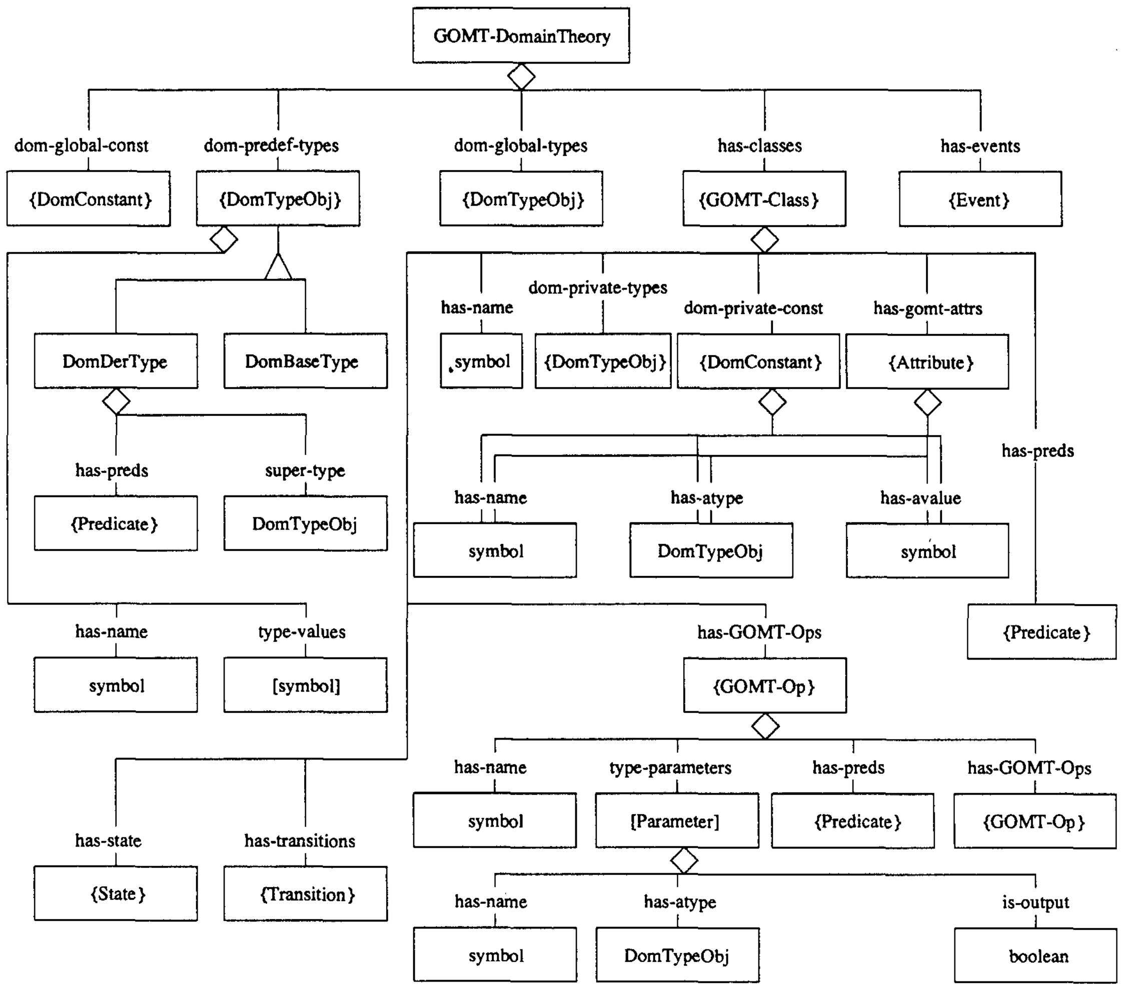

Figure 3.1 The Abstract Syntax Tree Used in the Domtree Software.

in the Elicitor Harvester. The structure of the tree, as well as the information within the tree, was examined to gather necessary requirements for the Elicitor Harvester design.

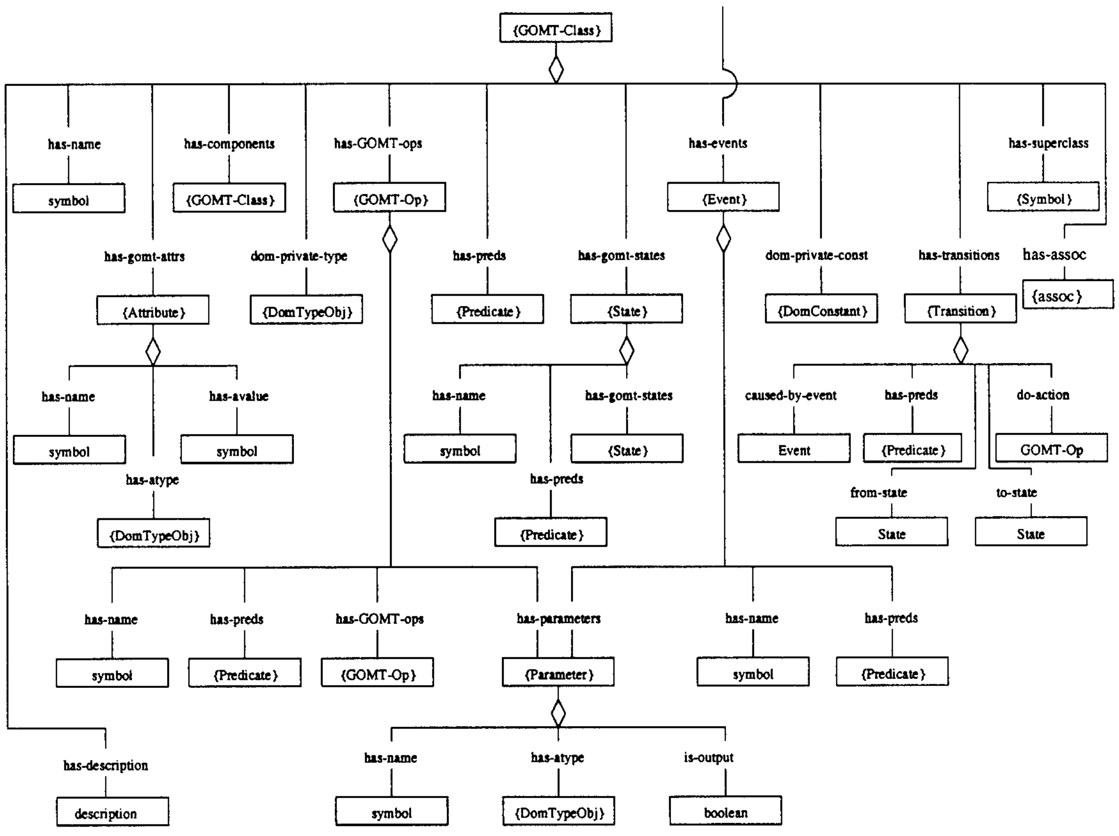

Figure 3.2 The Abstract Syntax Tree used in the Elicitor Harvester.

The tree in Figure 3.2 is based on the GOMT AST in Figure 3.1 with a few additions and alterations necessary for the the development of the Elicitor Harvester. Those changes include:

1. Addition of a description attribute to class - To provide the user of the Elicitor Harvester with a description of the individual objects within the domain, this field was added. The intent is to give the user a short one line description of each object class.

2. Addition of an association attribute to class - This was added to capture all associations of a particular object. It captures the associated object along with the cardinality information.

3. Identification of components within an aggregate - To avoid confusion between attributes and components, the Elicitor Harvester will consider all attributes that have the same attribute name and type name (i.e. jetengine of type Jetengine) as a component. This component will be represented separately from the attributes on the AST. The number of each component allowed within an aggregate are captured in the constraints (i.e. $1 \leq jetengine \leq 4$) of that aggregate.

3.2.2 Output of the Elicitor Harvester. For this phase in the Elicitor Harvester development, the output takes the form of a new AST based on the input AST. Since the output varies depending on the individual task at hand, it is a subset of the input AST. The output can then be used by any future automated system. Another possibility is the production of LaTeX Z specifications which may take the same form as the domtree input. However, at this time the goal is a correct and consistent AST.

3.3 Problem Forming - Gathering the Requirements

Now that decisions have been made about how the input to the Elicitor Harvester looks and what is expected as output, it is time to develop the requirements. These requirements are broken up into two levels, the high level requirements and the low level requirements.

3.3.1 High Level Requirements. Top level requirements are listed below and are the basis in developing the lower-level requirements. The goal of the Elicitor Harvester, hence the top level requirement, is the creation of a new aggregate based on the domain model. This results in three major requirements for the Elicitor Harvester.

1. Change Associations and Objects - The ability to alter specifications to suit the new system is required.

   (a) Add Attributes and Constraints - The need to add attributes and associated constraints to existing domain components may be necessary in the creation of new specifications.

   (b) Alter Constraints and Attributes - Constraints and attributes must be able to change. Constraints will only be allowed to become more restrictive.

   (c) Change Types - When using components, a need for type changes may occur to ensure consistency throughout the new system specifications.

   (d) Add Operations - New and existing operations are the cornerstone to the user interface approach to the Elicitor Harvester. As such, the ability to declare new operations, to include specifying input and output parameters, along with pre- and post-conditions, is essential to the success of the Elicitor Harvester. The importance of this is pointed out in Section 4.2.

   (e) Add Sub-States - Although states will not be allowed to be altered, sub-states may be added within a state and existing events may be used in the transition of the new sub-states.

2. Select Existing Objects - This requirement involves the selection process necessary to identify the proper objects for any given system specification.

3. Create New Aggregate - In order for the Elicitor Harvester to be able to change dynamically, the ability to derive new aggregates must be available. This feature allows for a building block approach in the development of new specifications.

3.3.2 Low Level Requirements. The main requirements, altering and choosing objects in the creation of new system specifications, have been mentioned above. However, many lower level requirements needed to be identified to ensure proper design of the Elicitor Harvester. The next few sub-sections discuss the process used in identifying those requirements.

The ability to alter specifications to fit a particular need is the key to the Elicitor Harvester. To do this, each item in the domain tree was to be looked at individually and a decision on which items could be changed, added, or deleted was made. This information is captured in Figure 3.3 The first column contains the name of the parts of the domain tree. The second, third, and fourth column contain an X if that item can be changed, added, or deleted. The fifth column indicates whether this version of the Elicitor Harvester deals with the specific part. The final column gives a brief description of each item.

The following list contains all of the domain tree attributes used in the Elicitor Harvester and identified in Figure 3.3. Brief justifications are given on why each attribute was allowed to be added, deleted, or changed. Simple examples are used for clarity when necessary.

Domain Tree

Domain Tree Item	Chnge	Add	Del	Subset	Brief Description
has-classes		X	X	X	Indicates list of classes contained in a domain
has-name		X		X	Indicates the name of the class
has-gomt-attrs	X	X	X	X	Indicates the attributes or components of a class
has-name		X			Indicates the name of each attribute or component
has-atype	X	X			Indicates the data type of the attribute or component
has-avalue		X			Indicates a specific value that an attribute contains
has-preds	X	X			Indicates constraints within the class
dom-private-types					Indicates any type that is unique to this class
dom-private-const					Indicates any constants unique to this class
has-GOMT-Ops		X			Indicates list of operations that an object can perform
has-name		X			Indicates the name of the operation to be performed
has-preds		X			Indicates the constraints (pre-post conditions) of a particular operation
has-parameters		X			Indicates the list of parameters for an operation
has-name		X			Indicates the name of the parameter
has-atype	X	X			Indicates the data type of the parameter
is-output		X			Indicates whether a parameter is an output of an operation
has-GOMT-Ops		X			Indicates a sub-operation and contains same items as previous GOMT-Op
has-superclass					Indicates the parent class of an object
has-description					Indicates a general purpose of the object
has-associations			X		Indicates related objects/classes
has-gomt-states					Indicates the states the object can be in
has-name					Indicates the name of the state
has-preds					indicates the constraints of a particular state
has-gomt-states		X			Indicates substates within a state and contains same items as above
has-transitions		X			Indicates the list of transitions that cause state changes
cause-by-event		X			Indicates event that triggers transition
has-preds		X			Indicates constraints of transition
from-state		X			Indicates state being left
to-state		X			Indicates state being entered due to trigger event
do-action		X			Indicates an operation that must be performed because of the transition
has-events					Indicates the list of events within a domain -at same level as GOMT-Class
has-name					Indicates the name of the event
has-preds					Indicates constraints on the event
has-parameters					Indicates the list of parameters for an event
has-name					Indicates the name of the parameter
has-atype	X				Indicates the specific type of the parameter
is-output					Indicates whether a parameter is an output parameter

Figure 3.3 Items of the tree that can be altered.

1. has-classes - Any class can be added to a model to cover an association or sub-class of an existing class. Classes may also be deleted if the new domain model does not require that class. All changes to a class will be added as a sub-class.

2. has-name - New sub-classes may be given new names. However, existing names will not be altered to avoid confusion.

3. has-gomt-attrs - New attributes can be added as long as they are necessary in the creation of a new class. Attributes may also be deleted if not pertinent to the new domain. This may occur if a constraint specifies a range of components starting at zero. For example, a $0 \leq throttle \leq 2$ constraint indicates that a throttle may be deleted. Changes may only occur within the type.

4. has-name - Names can be added only if the attribute is being added. Names may not be deleted or changed to avoid confusion.

5. has-atype - Types may be added to a new attribute. Types may also be changed to fix conflicts with related objects. For example, fueltank.weight and jetengine.weight may both need to be represented in pounds because their aggregate, propsystem, adds both together and stores the results in pounds.

6. has-avalue - Values may be added to new attributes. Any changes should not be considered because domain modelers must have a good reason to put specific values on attributes.

7. has preds - Predicates, or constraints, may be changed only to make them more restrictive (decrease ranges). Increasing ranges may violate a property of the original domain. For example $32 \leq watertemp \leq 212$ may exist in a domain because freezing

or boiling may cause problems. Predicates may also be added when creating a new class. Since constraints were probably put in for a good reason, deleting predicates will not be allowed.

8. domain-private-types - These types are there because they are domain specific and no change, addition, or deletion should be allowed.
9. domain private-const - These constants are also domain specific and should not be allowed to change. For example, freezing might be equal to 32 degrees and must not be changed.
10. has-GOMT-Ops - Operations will be allowed to be added to a class, but not altered. Chapter 4 discusses the reasoning behind this decision.
11. has-name - Names will be added for all new operations. No deletions or changes may be made to existing names.
12. has-preds - Predicates will be added for all new operations. No deletions or changes may be made to existing predicates.
13. has-parameters - Parameters may be added for all new operations. No deletions or changes may be made to existing predicates.
14. has-name - Parameter names may be added to all new operations. No deletions or changes may be made to existing parameter names.
15. has-atype - Parameter types may be added to all new operations. In addition, parameter types may change due to conflicts with other objects. No deletion may be performed on parameter types.

16. is-output - Parameters may be added as output to all new operations. No deletions or changes may be made to existing output parameters.

17. has-GOMT-Ops - Same as has-GOMT-Ops above.

18. has-superclass - No change, deletion, or addition may be made to superclass. The domain modeler set up the class hierarchy and it must not change.

19. has-description - This is entered by the domain modeler and must not be changed, added, or deleted.

20. has-associations - Associations may be deleted if the object associated with it is not needed for the new system. However, associations may not be changed because it captures pertinent information for the domain.

21. has-gomt-states - States may not be changed, added, or deleted since they are relevant to the domain. However, sub-states may be added within existing states to enhance the model.

22. has-name - No addition, deletion, or change of name is allowed with the exception of addition to capture a new sub-state.

23. has-preds - No addition, deletion, or change of predicates are allowed with the exception of additions for new sub-states.

24. has-gomt-states - Addition of new sub-states are allowed but must be fully contained within the parent state. Only existing events will be allowed when considering new sub-states.

25. has-transitions - Transitions may be added to handle any addition of new sub-states.

26. caused-by-event - The cause-by-event may be added for a new sub-state. However, the event must be an existing event.

27. has-preds - predicates may be added for the transition needed for the new sub-state.

28. from-state - A from state may be added, but it must be either the parent state, or the from state of the parent state.

29. to-state - The new sub-state will be added as the new to-state.

30. do-action - A do action may be added to the new sub-state.

31. has-events - Has-events is at the same level as has-classes, thus events may not be added, deleted, or changed. The Elicitor Harvester only looks at the has-classes and below. The only exception is any types that belong to the class will change if it is also used in the event.

32. has-name - No changes, additions, or deletions allowed. See has-events.

33. has-preds - No changes, additions, or deletions allowed. See has-events.

34. has-parameters - No changes, additions, or deletions allowed. See has-events.

35. has-name - No changes, additions, or deletions allowed. See has-events.

36. has-atype - Types may change only if the type has been changed within the class that owns the parameter. In other words, if the event has a parameter that belongs to a class that the Elicitor Harvester is altering, then the type will then change if it is necessary to correct conflicts within the domain.

37. is-output - No changes, additions, or deletions allowed. See has-events.

3.4 System Development Requirements

As stated earlier, the Elicitor Harvester is designed to aid in the process of incremental development. Since many features must be added over time, the design discussed in Chapter 4 must take into account the capability of design enhancements. This allows for the addition of artificial intelligence as well as different input and output formats. Another design requirement includes the user-friendliness of the interface. The final design requirement was that the system must be able to be implemented. These are discussed further in Chapter 4.

3.5 Conclusions

This chapter discussed the goals and requirements of the Elicitor Harvester. It covered some background information on OOA and ASTs and how they tie into the Elicitor Harvester requirements. Next, the input and output of the system was given. From there requirements were listed at two different levels, high and low. Finally, it discussed the requirements put on the development and design of the system.

IV. Design

4.1 Introduction

This Chapter focuses on the design of the Elicitor Harvester, which incorporates success and avoids failures of past efforts and allows for flexibility for future enhancements. Firstly, the approach used to arrive at the design is discussed. This includes alternative approaches with their advantages and disadvantages. Secondly, the overall design is examined, including the high level algorithm used to build the Elicitor Harvester. Thirdly, the modules within the high level design are explained. Finally, the conclusion points out the highlights of this chapter.

4.2 Approach

This section discusses the approach taken to the design of the Elicitor Harvester. First, the process of defining approaches is discussed. Then, the four major approaches are examined. These approaches take two different forms. The first two look at modifying the existing AST model, while the second two involve creating a new aggregate based on the input AST model. These approaches are evolved into the final design used for the Elicitor Harvester. Two different types of domain models were used to analyze the approaches and design of the system. They are the *school* and missile *domain* (see Appendix A). These example domains helped keep the focus on the exact problem at hand. A major difficulty in determining the approaches was keeping focused on a system that produces software specifications and not a solution to specific software requirement.

4.2.1 Thought Process. By using the two specific domain models mentioned above, two major types of domain models were able to be considered in the design. The school domain uses many associations. This made it possible to observe how associations affect the specification. The missile domain contained many aggregates and inheritance features, allowing for the examination of how depth of domains affect the specification. Concentrating on the two different features, associations and aggregates, allowed covering the majority of situations encountered when dealing with domain models.

To come up with design alternatives, it was sometimes useful to come up with a scenario to examine the effects of different choices. This was mostly necessary on the fourth approach and involved the school scenario as well as the missile scenario. Looking at differing scenarios allowed identifying the necessary pieces of information as well as generic approaches for gathering the information. With this in mind, the four approaches considered for the Elicitor Harvester design are presented.

4.2.2 First Approach. The first approach amounted to a brute-force method. Users would use the existing domain model and would have to answer questions on each and every item in the domain tree that is allowed to change. The user interface would inquire whether an item was correct or needed to be changed and would make the change to the existing domain tree. For example, the missile might have a field for maximum attachment weight of 2200 pounds. A user might want to change this to 1000 kilograms. This would happen for every attribute, component, operation, state, and transition in the missile domain.

This approach has a few advantages. First, the entire model would be covered for every specification. This allows for all parts to be examined by the user. Second, there is no need to construct a new AST for that domain which reduces the complexity of implementation immensely.

Many disadvantages exist for this approach. In the first place, the user would be bombarded with many choices, even when he may only be concerned about making a propulsion system. In the second place, the user must be very knowledgeable and consistent as he goes through the entire system, piece by piece. This pushes some of the responsibility of checking for conflicts onto the user. In the third place, the system would not take on a natural feeling in its exchange of information with the user. Finally, unnecessary questions would be asked. If a user is designing a propulsion system, many questions would be asked about the missile, airframe, and other parts of the missile.

4.2.3 Second Approach. The second approach is similar to the first in that it would alter the existing domain model. However, it would take an extra step and allow for the elimination of unnecessary parts when it is determined that those parts are unnecessary. Using the missile example once again, consider a user wanting to specify a missile using a rocket engine. Instead of asking the user about every single attribute in the throttle, the system would eliminate the throttle when it is determined that a throttle is not needed. The user interface would ask for the same type of information as in the first approach, but it would not ask about components when they are not needed in the above aggregate.

The major advantage of this approach is the reduction in the amount of unnecessary questions asked to the user. By eliminating parts, the Elicitor Harvester would skip the questions on unnecessary classes. This would reduce some of the reliance on the user's knowledge and memory as he traverses through the domain tree.

However, the disadvantage of an unnatural user interface would still exist. The domain tree could contain many unnecessary components that may have a null value. This would increase search time through the tree and keep many unnecessary leaves on the domain tree. Finally, the system may have to restore deleted components if it is later found out that the component is needed. For example, if the user picked a rocket engine and later determined that the missile needs to turn, the throttle would have to be added back into the propulsion system aggregate. Since the rocket engine does not require a throttle, the throttle would have been deleted when the rocket engine was selected. This could cause a great deal of excess work by deleting and adding components or associations.

4.2.4 Third Approach. The third approach would involve the construction of a new aggregate model, using the original domain model as a reference. The user interface would ask about the necessity of each class or object and a new aggregate would be built based on the user's decision. This approach would only ask about parts if they are needed and would take a building block concept. As pieces are needed, they would be added to the new domain tree while the original domain tree would not be altered.

Several advantages exist with this approach. First, the original domain tree would always be the same, so there is no possibility of losing constraints, component information, or association information. Second, as parts are needed, the user would be asked about them and only then would the parts be added to the new aggregate tree. This would cut down backtracking problems, since the new components would only be added when it is determined that the component is needed. Finally, the ability to build sub-classes is created without destroying the original class and its constraints. For example, if the school domain is being used and there is a need to create an honor student, the grade point average may have a new range of 3.0 to 4.0 and a probation student may be created using the constraints of 0.0 to 2.0. This constraint would not be possible in the first two approaches since the honor students would have been created on the original domain tree and the probation student would no longer fit in the new range of grade point averages.

There are some disadvantages to this approach. The first one is that it still relies on a user that is knowledgeable of the domain, since decisions on whether certain objects are needed are laid upon the user. The second disadvantage is that the user interface is still not that simple or natural because the process of going through the domain tree object by object requires that some objects that are not necessary would be displayed to the user for input. The final disadvantage is that the system would now keep track of two different domain trees, the original and the new aggregate. However, this is a tradeoff that is acceptable since it could reduce the amount of necessary backtracking.

4.2.5 Fourth Approach. The fourth approach would also involve the creation of a new aggregate but would take on a totally different approach with the user interface. The "operations" approach would call for the user to identify or create operations necessary for the system under design. For example, if the user is building a system using the school domain, he may want to create a scheduling system. If a scheduling operation exists, all components necessary for the operation would be gathered and placed in the new aggregate. All components added would be based on their need in the desired operations.

Many advantages beyond those for the third approach exist in this approach. First and foremost, the user interface would be very natural for the creation of new systems. Since the creation of new systems arise due to the need for desired functionality, the creation of new or modification of existing operations should be exactly what the user of the Elicitor Harvester has in mind when specifying the required system. Another advantage over the first three approaches is that only the components that are required for the functionality would be added to the tree and the user would be asked about no unneeded components. This would reduce the number of questions necessary for the construction of the new system specifications. Finally, the amount of domain knowledge required by the user would be reduced. Since the user needs to know only about the functions or operations necessary for his system, the need to know how other unnecessary objects affect the needed objects would be greatly reduced.

The main disadvantage to this approach is with the creation of new operations. Since the user may be creating new operations, he may not know all of the necessary components needed for that operation. This may also increase the need of the user to know how to specify pre- and post-conditions. Since some necessary information may be several associations away or several levels deep within aggregates, the entire big picture may be a little hard to capture within the mind of the user.

4.3 Design Concept

This goal of this section is to demonstrate the interaction desired with the Elicitor Harvester. The school domain from Appendix A will be used as the example domain and the creation of a new operation, ComputeProbationList, will be created. This list is intended to contain the names of all students with a grade point average below 2.0. The following is a demonstration of the creation of this new function:

```
The Attributes/Components of the School Domain are:
  Faculty - Instructors of courses
  Course - The courses offered
  Quarter - Time periods in which courses are offered
  Room - Rooms used for course offerings
  Section - An actual course offered during a quarter
  Book - The required books for a section
  GradClass - A group of students due to graduate at the same time
  Student - The people enrolled in the school
  Gradeable - A specific item being graded in a course
The Available Operations of the School are:
  Enroll - Enrolls a new student
  Register - Registers students for courses
Are any of the above operations suitable:  NO
Would you like to create a new operation with the above
                                  attributes/components:  YES
Enter the name of the new function:  ComputeProbationList
Enter the input parameter(s):  Student, Section
Enter the output parameter(s):  Student
Enter the pre-condition:  cardinality(Students) > 0
Enter the post-condition:  0.0 <= ((Sum(grade))/(Sum(section.hours))< 2.0
Entering the Build Portion of the Elicitor Harvester
Student is being added to the new domain model
  Is the Name attribute needed?:  YES
  Is the Address attribute needed?:  NO
  Is the Sex attribute needed?:  NO
  Is the Birthdate attribute needed?:  NO
Student has a member_of Association with GradClass.
                                      Is this needed?:  NO
Student has a r_advises Association with Faculty.
                                      Is this needed?:  NO
Student has a scored(Score) Association with Gradeable.
                                      Is this needed?:  NO
Student has a assigned(grade) Association with Section.
                                      Is this needed?:  YES
Section is being added to the new domain model
  Is the Number attribute needed?:  NO
  Is the Hours attribute needed?:  YES
Section has a text association with Book.
                                      Is this needed?:  NO
```

```
Section has a teaching association with Faculty.
                                        Is this needed?:  NO
Section has a graded_by association with Gradeable.
                                        Is this needed?:  NO
Section has a occupied association with Room.
                                        Is this needed?:  NO
Section has a taught_as(Offering) association with Course.
                                        Is this needed?:  NO
The New Domain Model is
School
  has attributes:  ProbationList : set(student)
  has components:  Student : student
                     has attributes:  Name
                   Section : section
                     has attributes:  Hours
  has operations:  ComputeProbationList
    has pre-condition:  cardinality(Students) > 0
    has post-condition:  0.0 <= ((Sum(grade))/(Sum(section.hours))< 2.0
End School Domain
```

This scenario demonstrates the desired interaction of the Elicitor Harvester. The results should be a new school domain which contains all necessary information.

4.4 Design

The overall design of the system was based mainly on the fourth approach mentioned above. However, many of the ideas mentioned in the first three approaches were also adapted. The diagram in Figure 4.1 shows the major modules used in the design of the Elicitor Harvester. The design is broken into two halves. The half above the build module, which is referred to as the upper half, contains the functionality associated mostly with approach four along with some of the ideas from approach three. The half including the build module and all modules below it, which is referred to as the lower half, contains the functionality contained in the first two approaches.

The reasoning behind this design is that it allows the advantages of all the approaches to be combined into a single system while eliminating some of the disadvantages of those approaches. By having the upper half eliminate all unnecessary components before going to the build stage, questions will be limited to components necessary for the operation being constructed or identified at that time. Then the lower half is able to do all of the detailed work required to completely specify the components necessary for that operation. The following two sections discuss the details of the modules included in the upper half and lower half.

Another important aspect of the Elicitor Harvester design was the use of recursion. Since the domain tree is an abstract syntax tree, a way of going through all necessary leaves on the tree was needed. To avoid the excessive use of pointers and awkward loops, recursion was used in two spots, the call back to ElicitorHarvester from NextObject and the call back to Build from DetermineNeed. This allows for easy flow through all of the nodes of the tree and is used in much the same way that compilers use recursion in the processing of their abstract syntax trees.

4.4.1 Upper Half Design. As stated earlier, the upper half is used to capture the objects needed for the specific operations required by the user. This operation can be an existing operation specified at the top level of the domain, an operation specified at a lower level component, or an operation that must be built from scratch. This approach allows for a generic approach to the Elicitor Harvester. The following upper half modules along with their associated explanations and pseudo code are:

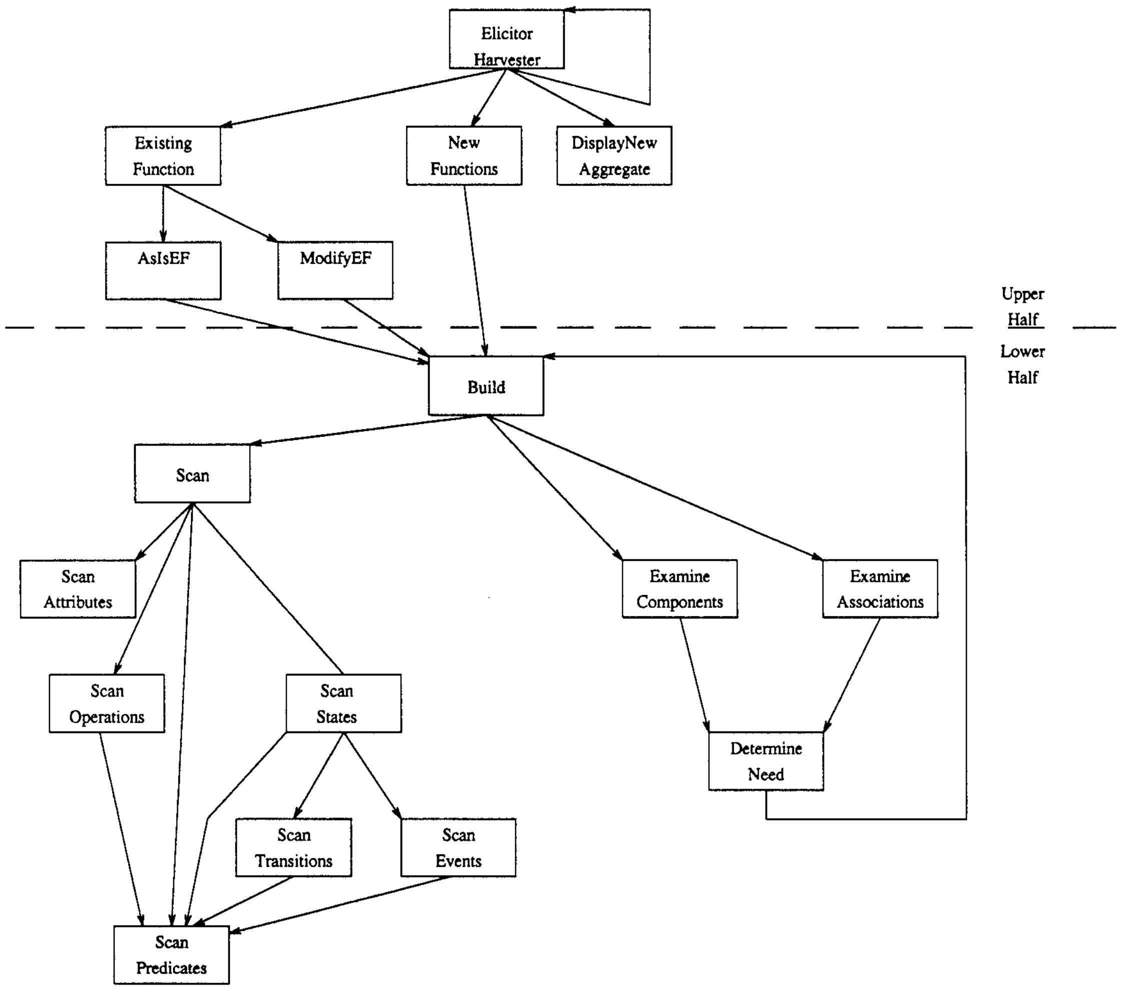

Figure 4.1 A High Level Design of the Elicitor Harvester

1. *ElicitorHarvester* - This module is the driver of the Elicitor Harvester and controls the generation of the components necessary for the existing and new operations. It also proceeds down the components to identify any existing operation at the lower levels.

```
Input:        GOMT-Class
Output:       GOMT-Class
Pseudo Code:
              loop through each existing GOMT-Op and prompt
                if GOMT-Op needed
                  call ExistingFunction(GOMT-Op)
                end if
              end loop
              loop through each existing GOMT-Class component
                if GOMT-Class needed
                  call ElicitorHarvester(GOMT-Class)
                end if
              end loop
              loop until finished
                if new function needed
                  call NewFunction(GOMT-Class)
                end if
              end loop
              DisplayNewAggregate(New-Class)
              Return GOMT-Class
```

2. *ExistingFunction* - This module will be called if the operation needed for the new specifications already exists. It will then find out if the operation needs to be altered.

```
Input:        GOMT-Op, GOMT-Class
Output:       GOMT-Class
Pseudo Code:
              List GOMT-Op inputs and outputs
              if inputs and outputs ok
                call AsIsEF(GOMT-Op)
              else
                call ModifyEF(GOMT-Op)
```

```
end if
return GOMT-Class
```

3. *AsIsEF* - This module will be called in the event that the chosen operation is suitable as is. The proper objects used by the operation will be added to the new aggregate and the proper attributes will be added to the current GOMT-Class. The Build function will then be called.

```
Input:          GOMT-Op, GOMT-Class
Output:         GOMT-Class
Pseudo Code:
                if has-parameters(GOMT-Op) is GOMT-Class
                  add GOMT-Class to new aggregate
                else
                  add GOMT-Attribute to current GOMT-Class
                end if
                call Build(GOMT-Class)
                return GOMT-Class
```

4. *ModifyEF* - This module will be called in the event that a chosen operation needs an alteration. After all necessary changes occur, such as addition or subtraction of parameters, GOMT-Class or predicates, the Build function will then be called.

```
Input:          GOMT-Op, GOMT-Class
Output:         GOMT-Class
Pseudo Code:
                loop through parameter list
                  if parameter is ok
                    if has-parameters(GOMT-Op) is GOMT-Class
                      add GOMT-Class to new aggregate
                    else
                      add GOMT-Attribute to current GOMT-Class
                    end if
                  end if
                end loop
```

```
loop through predicate list
  if predicate ok
    add predicate to has-preds
  else if predicate needs modification
    get new predicate
    add to predicate list
  end if
end loop
call Build(GOMT-Class)
Return GOMT-Class
```

5. *NewFunction* - In the event that an operation does not fit the user's need, this module allows for the user to identify input and output parameters, along with pre- and post-conditions for a new operation. After all necessary classes are added to the new aggregate, the Build function will then be called.

```
Input:        GOMT-Class
Output:       GOMT-Class
Pseudo Code:
              Get name of new operation and add to has-name(GOMT-Op)
              loop through GOMT-Attributes
                if input parameter
                  add to has-parameters(GOMT-Op)
                if output parameter
                  add to has-parameters(GOMT-Op)
                end if
              end loop
              loop through GOMT-Components
                if input parameter
                  add to has-parameters(GOMT-Op)
                  add to GOMT class as aggregate
                if output parameter
                  add to has-parameters(GOMT-Op)
                  add to GOMT class as aggregate
                end if
              end loop
              get input predicate and add to has-preds(GOMT-Op)
              get output predicate and add to has-preds(GOMT-Op)
              call Build(GOMT-Class)
              Return GOMT-Class
```

6. *DisplayNewAggregate* - After the entire system is specified, this module is responsible for printing out the new aggregate generated by the Elicitor Harvester.

```
Input:        GOMT-Class
Output:       N/A
Pseudo Code:
              loop through GOMT-Attributes
                print attribute
              end loop
              loop through GOMT-Components
                print Component
              end loop
              loop through GOMT-Ops
                print Op
              end loop
              loop through GOMT-States
                print State
              end loop
```

4.4.2 Lower Half Design. The lower half of the system is responsible for the generation of all necessary changes, additions, and deletions of the new aggregate. It implements many of the features identified in the first two approaches described above. The necessary classes are looked at one at a time and users are allowed to change parts of the specifications. The lower half is also responsible to traverse through aggregates and associations to explore whether they are needed for the operations being created. The following lower half modules include:

1. *Build* - This module is the entry point into the lower half. At this point, all classes necessary for the operation should be identified. Each class will be scanned. The components and associations will also be traversed.

```
Input:          GOMT-Class
Output:         GOMT-Class
Pseudo Code:
                Scan(GOMT-CLass)
                ExamineComponents(GOMT-CLass,GOMT-Comp)
                ExamineAssociations(GOMT-CLass,assoc)
                Return GOMT-Class
```

2. *Scan* - This module is responsible for sequencing the scan through all of the parts of the class. The operations, states, and transitions are scanned and all attributes needed for them will be added to the attribute list before being scanned. The transitions are scanned before states to ensure any state in a necessary transition is added to the list.

```
Input:          GOMT-Class
Output:         GOMT-Class
Pseudo Code:
                ScanOperations(GOMT-CLass,GOMT-Ops)
                ScanTransitions(GOMT-CLass,GOMT-Transitions)
                ScanStates(GOMT-CLass,GOMT-States)
                ScanAttributes(GOMT-CLass,GOMT-Attributes)
                Return GOMT-Class
```

3. *ScanAttributes* - This module is called to scan through all attributes within a class and allows for alteration of the attributes by the user. Only types and initial values may be altered.

```
Input:          GOMT-Class, GOMT-Attributes
Output:         GOMT-Class
Pseudo Code:
                loop through each attribute
                  if attribute type needs to be changed
                    change type
                  else if value needs to be changed
                  change value
                end if
              end loop
              Return GOMT-Class
```

4. *ScanOperations* - This module scans through all of the operation within a class and allows the user to validate the necessity of each operation. Any attribute or component needed for the operation will be added to the appropriate list.

```
Input:        GOMT-Class, GOMT-Op
Output:       GOMT-Class
Pseudo Code:
              loop through each operation
                loop through each parameter
                  if parameter not in attribute or component
                    add to appropriate place
                  end if
                end loop
                ScanPredicates(GOMT-Class, GOMT-Op)
              end loop
              Return GOMT-Class
```

5. *ScanPredicates* - This module is used to scan through predicates within a class. It includes predicates for constraints, pre- and post-conditions, and transitions. Users are asked to validate the correctness of the predicates.

```
Input:        GOMT-Class, Predicate
Output:       GOMT-Class
Pseudo Code:
              loop through each predicate
                for each attribute or component
                  if attribute not in class
                    add to appropriate place
                  end if
                end for
              end loop
              return GOMT-Class
```

6. *ScanStates* - This module is used to scan the states of a class and the user is asked to validate the correctness of the states, and asked if the need for sub-states exist. This is done after all states necessary for transactions are placed into the new aggregate.

```
Input:        GOMT-Class, State
Output:       GOMT-Class
Pseudo Code:
              loop through each state
                if state needed
                  ScanPredicates(GOMT-Class, Predicate)
                  loop through each sub-state
                    ScanPredicates(GOMT-Class, Predicate)
                  end loop
                  prompt user for sub-state addition
                  if sub-state addition
                    get state name and predicates
                    add to tree
                  end if
                end if
              end loop
```

7. *ScanTransitions* - The transitions are scanned by this module to ensure the accuracy of each transition.

```
Input:        GOMT-Class, Transition
Output:       GOMT-Class
Pseudo Code:
              loop through each transaction
                if transaction needed
                  if from state not yet in states
                     add from state to states
                  end if
                  if to state not yet in states
                    add to state to states
                  end if
                  ScanPredicates(GOMT-Class, Predicate)
                end if
              end loop
```

8. *ScanEvents* - This module scans the events reacted to by the class being explored. The user is asked to validate the correctness of these events. At this time, the event will only be displayed to the user. This decision was made because the goal of the Elicitor Harvester is to implement changes to GOMT-Class and objects below. Events are at the same level as GOMT-Classes.

```
Input:         GOMT-Class, Event
Output:        GOMT-Class
Pseudo Code:
               loop through events in domain theory
                 print event-name
               end loop
```

9. *ExamineComponents* - This module is used to traverse through the components within a class. The goal of this module is to identify components that may be necessary for the operation being developed. If the component is in the component list, it will call build to customize the object. Otherwise the user will be prompted to ensure the component is not needed at this time.

```
Input:         GOMT-Class, Component
Output:        GOMT-Class
Pseudo Code:
               loop through components in class
                 if component in the new class
                   Build(GOMT-Class)
                 else
                   DetermineNeed(component)
                 end if
               end loop
```

10. *ExamineAssociations* - This module is used to traverse through associated classes and helps in identifying the need of each associated class. This module does the same as ExamineComponents except it is performed on associations.

```
Input:         GOMT-Class, association
Output:        GOMT-Class
Pseudo Code:
               loop through associations in class
                 if associations in the new class or parent class
                   Build(GOMT-Class)
                 else
                   DetermineNeed(association)
                 end if
               end loop
```

11. *DetermineNeed* - This module is used by both ExamineComponents and ExamineAssociations and traverses through other associations and sub components by recursively calling build. No associated class or component will be deleted until all of its relationships are explored and the need of the class is determined. This is done to ensure that a class, either association or component, has all of their relationships explored before they are rejected.

```
Input:         GOMT-Class
Output:        GOMT-Class
Pseudo Code:
               loop through all associations in class
                 if associations is needed by user
                   Build(GOMT-Class)
                 end if
               end loop
               loop through all component in class
                 if component is needed by user
                   Build(GOMT-Class)
                 end if
               end loop
```

4.5 Conclusion

This chapter presents the detailed design used in the Elicitor Harvester. Through the use of two example domain models that are diverse in their configurations, one is flat with many associations while one is deep with many different levels of components, approaches were developed in a way that domain model diversity was addressed. The advantages and disadvantages of the four different approaches were discussed and a design was developed in a way that maximized those advantages while it minimizing those disadvantages. The major modules were then discussed along with the purpose of those modules. The next chapter discusses the implementation choices along with the testing strategy used in the validation of the system design.

V. Implementation and Testing

5.1 Introduction

This chapter discusses the details of how implementation and testing of the design discussed in Chapter 4 were handled. First, the implementation decisions, benefits and difficulties are discussed. This is followed by a discussion of the test strategy, test case development, and use of test data.

5.2 Implementation

Since demonstrating feasibility of the Elicitor Harvester was the main goal of this research, implementation of all functions identified in Chapter 3 was not necessary. However, the changes that were implemented needed to be complete and acceptable. To accomplish this, the system was developed incrementally. The following sub-sections discuss the strategies and difficulties encountered during implementation.

5.2.1 Implementation Strategy. Many choices were made to enhance the success of the implementation portion of the Elicitor Harvester. These choices include a language, supporting software, and methodology for implementation. The code for the Elicitor Harvester is contained in Appendix B.

The language chosen for the implementation of the Elicitor Harvester was Software Refinery version 4.0(Refine). There were two major factors that played a role in this selection. First and foremost was the ability of the language to build, manipulate, and traverse abstract syntax trees. Since the domain model is an AST, the power and built in features offered by Refine greatly reduces the amount of code necessary for implementation.

The second reason for choosing Refine was the availability of supporting software used in conjunction with the Elicitor Harvester. Those include the Inspector software which is a commercial product packaged with Refine, and the domtree (5) and Z parsing software (11), which are all explained in the next paragraph.

The majority of the software used in support of the implementation of the Elicitor Harvester was written at AFIT. This software was used for front end processing and testing. The purpose of the Z parsing software is to simply parse Z specifications in from a file into a unified Z AST. However, this software lacked the ability to read in descriptions needed for aiding the selection process as well as the ability to differentiate between attributes and components. The purpose of the domtree software is to handle the transformation from the Z AST into the DOM AST, which is reflected in Figure 3.1. There was still a need to alter the DOM AST to capture associations, descriptions and components as mentioned in Section 3.2.1. Inspector software was also used in support of testing. This software is a tool included with Refine and was a helpful aid in testing the Elicitor Harvester. By providing the capability of viewing and tracking changes made to the ASTs, debugging time and effort were greatly reduced.

The methodology for implementation was incremental. Each of the major modules identified in the design was implemented as a stub. Functionality was then added to these stubs until the allotted time ran out. The reasoning behind this approach was that feasibility was the major goal, and by constructing a system that was able to be altered in an incremental manner an evaluation of individual functions was possible.

5.2.2 Implementation Difficulties. The major difficulty involved with implementation was the Refine learning curve. Although Refine is an excellent language for manipulating ASTs, it is not a typical programming language. Structures such as looping and decisions were a little more difficult than in conventional languages. However, user interaction was, by far, the most difficult aspect to handle. User interaction was a major problem. Refine sits atop the lisp environment and uses pre-written input output routines within the lisp environment. This severely limited the use of user interface. Because of this, a command line interface was used and much of data displayed to the user appeared in a less than desirable format. The user interface as implemented is basic and may lack effective communication with the user. However, development of a good user interface was beyond the scope of demonstrating Elicitor Harvester feasibility.

5.3 Testing

The aim of testing was to demonstrate the feasibility of the Elicitor Harvester across diverse domains. The focus was on demonstrating that the generic approach taken was a promising one. To do this, a test plan was produced and test cases were developed to support the plan.

5.3.1 Test Plan. The test plan was developed to isolate the items identified in the subset column of Figure 3.2. Separate tests were developed for each of the items and Appendix C contains Figure 3.2 with an extra character next to each identified domain tree item. 'Y' indicates a successful test, 'N' indicates an unsuccessful test, and '-' indicates no test was performed. The ability of changing specifications up and down the hierarchy of classes was not necessary at this time. However, the ability of the system to properly alter individual specifications needed to be demonstrated.

Each test in the plan is accompanied by the criteria for a successful test as well as a simple example. The criteria for measuring success was to show the objects in their final state and to insure all necessary alterations had occurred. Final object information was compared to the expected results from the test plan. Any differences were recorded and examined for possible causes of the differences. Then simple changes were made, while design flaws were corrected and re-implemented when possible. Example output from the test is contained in Appendix D. This example of the counter object includes a scenario where an original operation is kept as is, an original operation is altered, and a new operation is created.

To accomplish the tests, the domtree software was used to load the specifications into the Elicitor Harvester AST, and that AST was examined with the inspector software to ensure proper loading. Then the test plan was executed case by case. The result of each test was examined using the inspector.

5.3.2 Test Cases. Small domain models were designed to test the system's ability to change all related parts. These domain models include the school and missile domains. To limit the number of objects, only portions of the two domains were used. Appendix E contains the Z specifications used for the tests.

The Z specifications used are from the AFIT library. These specifications required modification to allow for parsing by the Z parser. Since this was a time consuming process, modules were selected for testing based on their diversity. In other words, individual classes were picked based on the number of test cases that could be accomplished by its inclusion.

5.4 Results

The tests indicate that many of the specified low level requirements listed in Section 3.3 are indeed possible. The Elicitor Harvester is capable of traversing the AST and pulling required information to construct a modified domain model. The interaction with the user, including sequence of execution and building of GOMT Operations, aids in the construction of individual requirements. However, some missing requirements and flaws exist in the system. First, the build portion of the design needs to be re-structured to take advantage of redundant tasks. Second, the ability to separate attributes from components was not implemented. Third, the associations of each class are not identified. The second and third flaw are contributed to pre-processing of the specifications and indicate the need for the Elicitor Harvester to include a module to perform that pre-processing. Finally, the ability to examine constraints was not implemented. The goal was to check simple addition constraints in the following form:

```
attribute = componentA.attribute + componentB.attribute
```

This ability was needed to aid in the selection of components within a domain.

5.5 Conclusion

This chapter was meant to give the highlights of the implementation and testing of the Elicitor Harvester. This included the decisions, difficulties, and strategies used to reach a decision on the fate of the concepts and design of the system. It also pointed out the major results of testing. The next chapter contains that decision as well as an analysis of the results and suggestions for the future.

VI. Conclusions and Recommendations

6.1 Introduction

This final chapter discusses the success of the Elicitor Harvester, as well as its failures. First, conclusions on the success of the goals will be drawn. Those conclusions are based on the test results covered in Section 5.4. Those conclusions contain failures as well as successes. Finally, recommendations for future research in the area of the Elicitor Harvester are made.

6.2 Elicitor Harvester Conclusions

This research effort into the feasibility of an Elicitor Harvester was a success. The three major goals; defining the requirements, designing the system and demonstrating feasibility have been demonstrated. These three areas are expanded on in the next three sub-sections.

6.2.1 Conclusions on Requirements. Requirements from Section 3.3 are sufficient to build the Elicitor Harvester. In general, the requirements cover construction of an object-oriented Elicitor Harvester system. However, further refinement would be necessary for Items 2 and 3 in Section 3.3.1. This is discussed further in regards to implementation.

6.2.2 Conclusions on Design. The design of the Elicitor Harvester met the need for adaptability. Structurally, the top half of the design, indicated in Figure 4.1, is very sound and requires no changes. On the other hand, the lower half evolved during the implementation phase. Some of this evolution was due to language limitations, while others were necessary for correctness. Finally, the original design called for a table to keep track of attribute information such as those used in compiler construction. This detail was dropped from the original design and should be replaced. This table would be helpful for cross checking as well as compatibility verification.

6.2.3 Conclusions on Implementation and Feasibility. Implementation conclusions are drawn with respect to how they meet requirements and demonstrate feasibility. The high level requirements from Section 3.3.1 are able to be met. As mentioned above, much refinement is necessary on Item 2, which is the selection of existing objects. Item 3, the creation of new aggregates, is perhaps the most successful. However, much work needs to be done in the area of verification of the created aggregate. Finally, Item 1, the ability to change associations and objects, involves many of the low level requirements from Section 3.3.2. These requirements were met in part, but much must be done to fine-tune and expand the implementation. Although the implementation led to a limited demonstration of feasibility, none of the low level requirements were not met due to infeasibility.

6.3 Future Recommendations

There are several possibilities for future development of the Elicitor Harvester. The first involves the development of a domain specific user interface language and how it would increase usefulness. The second involves the incorporation of artificial intelligence and how it would enhance the existing system. The third involves the capabilities added with the use of a front end processor and how it may aid in some of the problem areas identified in Section 5.4. These three recommendations are covered in more detail in the following sub-section.

6.3.1 Develop Domain Specific User Interface Language. The most useful future area of research on the Elicitor Harvester would be in the area of user interfaces. During this research, much of the information needed required the user to be familiar with software development techniques and formal languages. However, for the Elicitor Harvester to reach larger groups of people, a domain specific language should be developed. This would allow domain experts, rather than software developers, to develop systems as needed. By implementing such a language, the positive impact of the Elicitor Harvester would greatly increase.

6.3.2 Incorporation of Artificial Intelligence. Perhaps the biggest shortfall of the Elicitor Harvester is the selection of objects in the construction of the system. Although this can be done under tight guidance criteria, a more probabilistic approach would aid immensely in the process of selection when dealing with an increasing domain size. This enhancement would aid the process of constructing the operation and in the selection of components.

6.3.3 Elicitor Harvester Front End Processors. The Elicitor Harvester may be able to adapt to several different formal languages. A key to this capability would be different parsers for various formal specification languages. At AFIT, the capability to parse Z and Larch already exist. Earlier parts of the research demonstrated the capability to parse Z directly into the GOMT AST. By using the AST from the Elicitor Harvester as a baseline, many other parsers may be designed to read specifications directly into the Elicitor Harvester. This has three advantages for the evolution of the Elicitor Harvester. Firstly, the system could parse files in as needed which adds efficiency to the system. Secondly, formal software specifications can be mingled, which increases the number of domains available. Finally, the problem of converting attributes into components is avoided by directly loading them into the proper location of the AST.

Bibliography

1. Al-Yasiri, Adil et-al. "Developing Software Systems with Domain Oriented Reuse," *IEEE*, 133–139 (Apr 1994).
2. Batory, Don et-al. "Domain Modeling in Engineering Computer-Based Systems." *Proceedings of the 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems*. 19–34. 1995.
3. Beem, Charles G. *Extending the Formal Object Transformation Process to Support Algebraically-Based Design Refinement: The Larch Perspective*. MS thesis, Air Force Institute of Technology, 1995.
4. Calmet, J. et-al. "Building Bridges Between Knowledge Representation and Algegraic Specifications." *Methodologies for Intelligent Systems: 8th International Symposium, ISMIS*. October 1994.
5. Hartrum, Thomas C. "An Object Oriented Formal Transormation System." draft Apr 96.
6. Owens, Ronald L. "Development and Implementation of a Domain-Specific Reuse Plan." *Tenth Annual Washington Ada Symposium Proceedings*. 31–42. 1993.
7. Petro, J. et-al. "Model-Based Reuse Repositories - Concepts and Experience." *Seventh International Workshop on Computer-Aided Software Engineering*. 60–69. 1995.
8. Prasad, Aarthi, et-al. "Reuse System: An Artificial Intelligence-Based Approach," *Journal of Systems Software*, *27*:207–221 (1994).
9. Pressman, Roger S. *Software Engineering A Practitioner's Approach*. McGraw-Hill Inc., 1992.
10. Simos, Mark A. "Juggling in Free Fall: Uncertainty Management Aspects of Domain Analysis Methods." *Advances in Intelligent Computin-IPMU '94: 5th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*. 512–521. 1995.
11. Wabiszewski, Kathleen M. *Unification of Larch and Z-Based Object Models to Support Algebraically-Based Design Refinement: The Z Perspective*. MS thesis, Air Force Institute of Technology, 1994. DTIC No. AD-A28923.
12. Ward, Martin P. "Language-Oriented Programming," *Software Components and Tools*, *15*(4):147–161 (1994).
13. Warner, Russell Mark. *A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System*. MS thesis, Air Force Institute of Technology, 1995. DTIC No. AD-A274128.
14. Whittle, Ben. "Models and Languages for Component Description and Reuse," *Software Engineering Notes*, *20*(2):76–87 (January 1995).

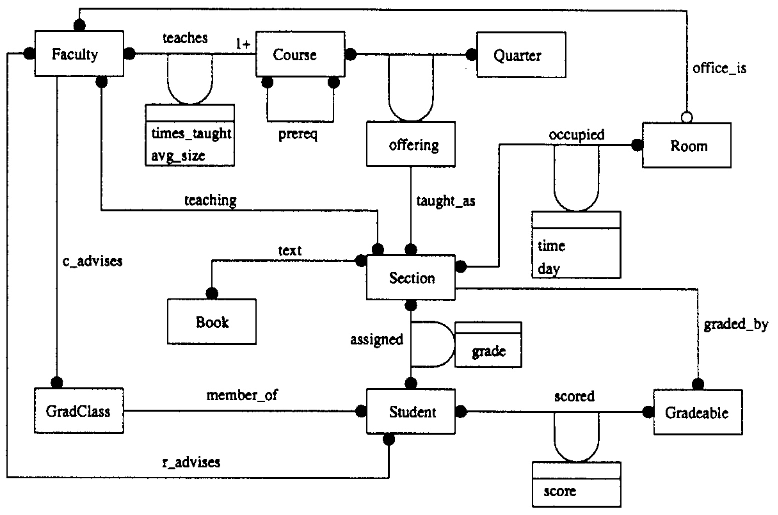

Figure A.1 Rumbaugh Model of the School Domain

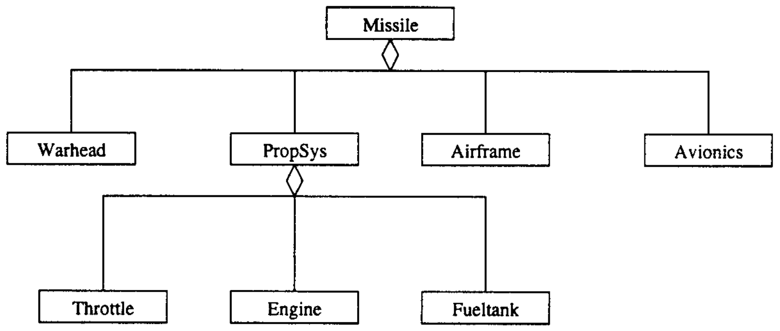

Figure A.2 Rumbaugh Model of the Missile Domain

Appendix B. Elicitor Harvester Code

```
!! in-package ("RU")
!! in-grammar ('user)

% Author: Capt Tim Karagias
% Filename: el-ha.re
% Description: This is the Elicitor Harvester system.
%
%

% Extensions to domtree
var Component      : object-class subtype-of Obj-Object
var has-superclass : map (GOMT-Class, symbol) = {||}
var has-components : map (GOMT-Class, set(GOMT-Class))
                         computed-using has-components(x) = {}
%%var has-description : map (GOMT-Class, string) = [""]
var has-description : map (GOMT-Class, symbol) = {||}

var eh-dom: GOMT-DomainTheory =
            make-object('GOMT-DomainTheory)
var eh-zed: DomainTheory = undefined

form Define-Elicitor-Harvester-Tree-Extensions

Define-Tree-Attributes('GOMT-Class,
                        {'has-name, 'dom-private-types,
                         'dom-private-const, 'has-gomt-attrs,
                         'has-components, 'has-description, 'has-preds,
                         'has-GOMT-Ops, 'has-gomt-states,
                         'has-transitions, 'has-parents, 'has-specials,
                         'has-superclasses, 'has-components,
                         'has-constraints, 'has-functionals})
%Define-Tree-Attributes('Component,
%                        {'has-name, 'has-atype, 'has-avalue, 'has-datatype,
%                         'has-initvalue}) &


% This Function calls the necessary software to load the domain tree
% and starts the Elicitor Harvester
function Prime(): GOMT-Class =
%function Prime() =
  Let (the-class: GOMT-Class = undefined)
  Let (eh-class: GOMT-Class = undefined)
  LoadMyZed();
  ZedToGomt(my-zed, eh-dom);
  Set-All-Types(eh-dom);
  ListDomain(eh-dom);
  (enumerate a-class over
    has-classes(eh-dom) do
      the-class <- a-class;
    eh-class <- ElicitorHarvester(the-class));
  eh-class


% Main driver, Responsible for gathering necessary information to
% build new operations
function ElicitorHarvester(a-class : GOMT-Class): GOMT-Class =
  Let (newclass: GOMT-Class = undefined)
  Let (op-name: symbol = 'Unk)
  Let (choice : boolean = False)
  Let (finished? : boolean = False)
  Let (newop? : boolean = False)
  Let (comp-name : symbol = 'Unk)
  Let (attr-name : symbol = 'Unk)

  newclass <- make-object('GOMT-Class);
  (enumerate an-op over
    has-gomt-ops(a-class) do
      op-name <- has-name(an-op);
      format(T, "Operation is ~D", op-name);
      choice <- Read-Boolean(" Do you need this operation?");
      format(T, "~%");
      choice = true --> newclass <- ExistingFunction(an-op,newclass)
  );
  (enumerate a-comp over
    has-components(a-class) do
      comp-name <- has-name(a-comp);
      format(T, "Component  ~D", comp-name);
      format(T, " does ");
      DisplayDescription(a-comp);
      choice <-
        Read-Boolean("Do you need to look at this component");
      format(T, "~%");
      choice = true --> newclass <- ElicitorHarvester(a-comp)
  );
  format (T, "Looking at ~D~%", has-name(a-class));
  (enumerate an-attr over
    has-gomt-attrs(a-class) do
      attr-name <- has-name(an-attr);
      format(T, "Attribute  ~D", attr-name);
      format(T, " does ");
      DisplayDescription(an-attr)
  );
  finished? <- false;
  newop? <- false;
  (while ~finished? do
    newop? <-
      Read-Boolean("Do you want a new operation using the above objects?");
    newop? = true --> newclass <- NewFunction(a-class);
    finished? <- Read-Boolean("Are you done? ")
  );
%  DisplayNewAggregate(newclass);
  newclass


% This function displays the description of an object to the screen
function DisplayDescription(a-class : GOMT-Class) =
  format(T," which ~D~%", has-description(a-class));
  format(T,"~%")


% This function checks the suitability of an existing function.  It
% calls one of two functions; the ModifyEF if function needs to
% change or AsIsEF if the function is ok the way it is
function ExistingFunction(the-op : GOMT-Op, a-class : GOMT-Class) : GOMT-Class =
  Let (choice : boolean = False)
  Let (newclass: GOMT-Class = undefined)
  format(true, "~/pp/", the-op);
  choice <- Read-Boolean( "Is the above operation ok as is~%");
  choice = true --> newclass <- AsIsEF(the-op,a-class);
  choice = false --> newclass <- ModifyEF(a-class,the-op);
  newclass

% This function builds a new function by asking for parameters, pre
% conditions, and post conditions.
function NewFunction(a-class : GOMT-Class) : GOMT-Class =
  Let (new-op : GOMT-Op  = make-object('GOMT-Op))
  Let (param-set: seq(parameter) = [])
  Let (new-param : parameter = make-object('parameter))
  Let (new-func-name : symbol = 'Unk)
  Let (finished? : boolean = False)
  Let (choice-in : boolean = False)
  Let (choice-out : boolean = False)
  Let (pre-cond : string = " ")
  Let (post-cond : string = " ")
%  new-op <- make-object('GOMT-Op);
  new-func-name <-
    Read-Symbol("Give the name of the new function: ");
%build tree here
  set-attrs(new-op,
          'has-name, new-func-name);
  (enumerate an-attr over
    has-gomt-attrs(a-class) do
      format(T, " The attribute is ~D~%", has-name(an-attr));
      choice-in <- Read-Boolean("Is this an input parameter? ~%");
      choice-out <- Read-Boolean("Is this an output parameter? ~%");
%      choice-in --> set-attrs(make-object('parameter),
      choice-in --> set-attrs(new-param,
              'has-name, has-name(an-attr),
                 'is-output, choice-out);
      format(T, " New Parameter : ~D~%", new-param);
      format(T, " New Parameter List: ~D~%", param-set);
      append(param-set, new-param)
% build tree here
  );
      set-attrs(new-op,
                 'has-parameters, param-set);
  (enumerate a-comp over
    has-components(a-class) do
      format(T, " The component is ~D~%", has-name(a-comp));
      choice-in <- Read-Boolean("Is this an input parameter? ~%");
      choice-out <- Read-Boolean("Is this an output parameter? ~%");
      choice-in --> set-attrs(make-object('parameter),
                 'has-name, 'an-attr,
                 'is-output, 'choice-out)
% build tree here
  );
  pre-cond <-
    Read-String("Give the pre condition: ");
  post-cond <-
    Read-String("Give the post condition: ");
% build tree here
  format(T,"New Operation: ~D~%", new-op);
  a-class

% This function displays the current GOMT Class
function DisplayNewAggregate(a-class : GOMT-Class) =
  (enumerate an-attr over
    has-gomt-attrs(a-class) do
      format(T, "attribute ~D", has-name(an-attr))
  );
  (enumerate a-comp over
    has-components(a-class) do
      format(T, "component ~D", has-name(a-comp))
  );
  (enumerate an-op over
    has-gomt-ops(a-class) do
      format(T, "Operation ~D", has-name(an-op))
  );
  (enumerate a-state over
    has-gomt-states(a-class) do
      format(T, "State ~D", has-name(a-state))
  );
  format(T,"in DisplayNewAggregate ~%")

% This function builds the existing function on the new Tree
function AsIsEF(a-class : GOMT-Class, the-op : GOMT-Op) : GOMT-Class =
  Let (new-op : GOMT-Op = make-object('GOMT-Op))

  set-attrs(new-op,
            'has-name, has-name(the-op),
            'has-parameters, has-parameters(the-op),
            'has-preds, has-preds(the-op));
  format(T, "Old Operation is: ~D~%", new-op);
  format(T,"in AsIsEF ~%")

% This function allows an existing function parameters and
%conditions to be changed.
function ModifyEF(a-class : GOMT-Class, the-op : GOMT-Op) : GOMT-Class =
  Let (newclass: GOMT-Class = undefined)
  Let (choice-in : boolean = False)
  Let (choice-out : boolean = False)
  Let (choice-type : boolean = False)
  Let (choice : boolean = False)
  Let (new-type : string = " ")
  (enumerate a-param over
    has-parameters(the-op) do
      format(T, "Do you need the parameter ~D", has-name(a-param));
      choice-in <- Read-Boolean("Is this an input parameter? ~%");
      choice-out <- Read-Boolean("Is this an output parameter? ~%");
      choice-type <- Read-Boolean("Is the Type OK? ~%");
      ~choice-type --> new-type <- Read-String("Please Enter New Type:  ")
% build tree here
  );
  (enumerate an-attr over
    has-gomt-attrs(a-class) do
% put in check to see if already parameter
      format(T, "Do you need the attribute ~D", has-name(an-attr));
      choice-in <- Read-Boolean("Is this an input parameter? ~%");
      choice-out <- Read-Boolean("Is this an output parameter? ~%")
% build tree here
  );
  (enumerate a-comp over
    has-components(a-class) do
% put in check to see if already parameter
      format(T, "Do you need the component ~D", has-name(a-comp));
      choice-in <- Read-Boolean("Is this an input parameter? ~%");
      choice-out <- Read-Boolean("Is this an output parameter? ~%")
% build tree here
  );

  (enumerate a-pred over
    has-preds(the-op) do
      format(T, "The current condition is: ~D ~%", a-pred);
      choice <- format("Do you need this condition ~D", a-pred)
% build tree here
  );
  format(T,"in ModifyEF ~%");
  newclass

% This function scans the original and new tree to ensure all
% necessary objects are included in the new tree.
function Build(a-class : GOMT-Class, the-op : GOMT-Op) : GOMT-Class =
  Let (newclass: GOMT-Class = undefined)
  Scan(newclass);
  ExamineComponents();
  ExamineAssociates();
  format(T,"in Build ~%");
  newclass

function Scan(a-class : GOMT-Class) =
  ScanOperations();
  ScanTransitions();
  ScanStates();
  ScanAttributes();
  format(T,"in Scan ~%")

function ScanAttributes() =
  format(T,"in ScanAttributes ~%")

function ScanOperations() =
  format(T,"in ScanOperations ~%")

function ScanPredicates() =
  format(T,"in ScanPredicates ~%")

function ScanStates() =
  format(T,"in ScanStates ~%")

function ScanTransitions() =
  format(T,"in ScanTransitions ~%")

function ScanEvents() =
  format(T,"in ScanEvents ~%")

function ExamineComponents() =
  format(T,"in ExamineComponents ~%")

function ExamineAssociates() =
  format(T,"in ExamineAssociates ~%")

function DetermineNeed() =
  format(T,"in DetermineNeed ~%")
```

Domain Tree

Domain Tree Item	Chnge	Add	Del	Subset	Brief Description
has-classes		X -	X -	X	Indicates list of classes contained in a domain
has-name		X -		X	Indicates the name of the class
has-gomt-attrs	X Y	X -	X Y	X	Indicates the attributes or components of a class
has-name		X -			Indicates the name of each attribute or component
has-atype	X Y	X -			Indicates the data type of the attribute or component
has-avalue		X -			Indicates a specific value that an attribute contains
has-preds	X -	X -			Indicates constraints within the class
dom-private-types					Indicates any type that is unique to this class
dom-private-const					Indicates any constants unique to this class
has-GOMT-Ops		X Y			Indicates list of operations that an object can perform
has-name		X Y			Indicates the name of the operation to be performed
has-preds		X Y			Indicates the constraints (pre-post conditions) of a particular operation
has-parameters		X Y			Indicates the list of parameters for an operation
has-name		X Y			Indicates the name of the parameter
has-atype	X	X Y			Indicates the data type of the parameter
is-output		X Y			Indicates whether a parameter is an output of an operation
has-GOMT-Ops		X -			Indicates a sub-operation and contains same items as previous GOMT-Op
has-superclass					Indicates the parent class of an object
has-description					Indicates a general purpose of the object
has-associations			X _		Indicates related objects/classes
has-gomt-states					Indicates the states the object can be in
has-name					Indicates the name of the state
has-preds					indicates the constraints of a particular state
has-gomt-states		X -			Indicates substates within a state and contains same items as above
has-transitions		X -			Indicates the list of transitions that cause state changes
cause-by-event		X -			Indicates event that triggers transition
has-preds		X -			Indicates constraints of transition
from-state		X -			Indicates state being left
to-state		X -			Indicates state being entered due to trigger event
do-action		X -			Indicates an operation that must be performed because of the transition
has-events					Indicates the list of events within a domain -at same level as GOMT-Class
has-name					Indicates the name of the event
has-preds					Indicates constraints on the event
has-parameters					Indicates the list of parameters for an event
has-name					Indicates the name of the parameter
has-atype	X Y				Indicates the specific type of the parameter
is-output					Indicates whether a parameter is an output parameter

Figure C.1 Y=Tested Successfully, N=Tested Unsuccessfully, -=Not Tested/Implemented

Appendix D. Example Output

```
dribbling to file "/home/hawkeye5/96d/tkaragia/timthesis/domtree/out1"

NIL
.> (prime)
Creating GOMT Class Counter
  Creating Attributes for class Counter
    Creating Attribute Value: My-Nat
    Creating Attribute Limit: My-Nat
Creating State InitCounter for class Counter
Creating Operation Increment for class Counter
Creating Operation Add for class Counter
        Parameter: AddValue: My-Nat (input)
        Parameter: NewValue: My-Nat (output)
Creating Operation Subtract for class Counter
        Parameter: DecValue: My-Nat (input)
        Parameter: NewValue: My-Nat (output)
=====================================================
                 DOMAIN: *UNDEFINED*
=====================================================
Global Types:
    ----------------------------------------
Global Constants:
    ----------------------------------------
Events:
    ----------------------------------------
Domain Classes:
  CLASS: Counter:
  SuperClasses:
  Local Types:
  Local Constants:
  Local Attributes:
    ATTR: Value: not defined!; No initial value!
    ATTR: Limit: not defined!; No initial value!
  Inherited Attributes: None.
  Constraints:
    Value<=Limit
  States:
    InitCounter
     Predicates:
      Value=0
      Limit=100

  Operations:
    Increment (IN ;OUT )
     Predicates:
      Value<Limit
    Add (IN AddValue ;OUT NewValue )
     Predicates:
      (#1<an ADDITION-EXPR>) <=Limit
    Subtract (IN DecValue ;OUT NewValue )
     Predicates:
      (#2<a SUBTRACTION-EXPR>) >=0
```

```
State Transition Table:
End of Table -----
  ----------------------------------------
CLASS: Counter:
SuperClasses:
Local Types:
Local Constants:
Local Attributes:
  ATTR: Value: not defined!; No initial value!
  ATTR: Limit: not defined!; No initial value!
Inherited Attributes: None.
Constraints:
  Value<=Limit
States:
  InitCounter
   Predicates:
    Value=0
    Limit=100

Operations:
  Increment (IN ;OUT )
   Predicates:
    Value<Limit
  Add (IN AddValue ;OUT NewValue )
   Predicates:
    (#8<an ADDITION-EXPR>) <=Limit
  Subtract (IN DecValue ;OUT NewValue )
   Predicates:
    (#9<a SUBTRACTION-EXPR>) >=0
State Transition Table:
End of Table -----
  ----------------------------------------
CLASS: Counter:
SuperClasses:
Local Types:
Local Constants:
Local Attributes:
  ATTR: Value: not defined!; No initial value!
  ATTR: Limit: not defined!; No initial value!
Inherited Attributes: None.
Constraints:
  Value<=Limit
States:
  InitCounter
   Predicates:
    Value=0
    Limit=100

Operations:
  Increment (IN ;OUT )
   Predicates:
    Value<Limit
  Add (IN AddValue ;OUT NewValue )
   Predicates:
    (#11<an ADDITION-EXPR>) <=Limit
  Subtract (IN DecValue ;OUT NewValue )
```

```
     Predicates:
      (#12<a SUBTRACTION-EXPR>) >=0
  State Transition Table:
  End of Table -----
    ---------------------------------------
====================================================
Operation is Increment Do you need this operation?: (T/t for true, F/f for false): T

##r GOMT
  Increment abstract?: RE::*UNDEFINED* class op?:
    RE::*UNDEFINED*
Is the above operation ok as is~%: (T/t for true, F/f for false): T
Old Operation is: #13<a GOMT-OP>
in AsIsEF
Operation is Add Do you need this operation?: (T/t for true, F/f for false): T

##r GOMT
  Add ( AddValue: My-Nat, NewValue: My-Nat) abstract?:
    RE::*UNDEFINED* class op?: RE::*UNDEFINED*
Is the above operation ok as is~%: (T/t for true, F/f for false): f
Do you need the parameter AddValue
Is this an input parameter? ~%: (T/t for true, F/f for false): t
Is this an output parameter? ~%: (T/t for true, F/f for false): t
Is the Type OK? ~%: (T/t for true, F/f for false): f
Please Enter New Type:  : real
Do you need the parameter NewValue
Is this an input parameter? ~%: (T/t for true, F/f for false): t
Is this an output parameter? ~%: (T/t for true, F/f for false): t
Is the Type OK? ~%: (T/t for true, F/f for false): t
Operation is Subtract Do you need this operation?: (T/t for true, F/f for false): f

Looking at Counter
Attribute  Value does  which *UNDEFINED*

Attribute  Limit does  which *UNDEFINED*

Do you want a new operation using the above objects?: (T/t for true, F/f for false): t
Give the name of the new function: : countbackwards
 The attribute is Value
Is this an input parameter? ~%: (T/t for true, F/f for false): t
Is this an output parameter? ~%: (T/t for true, F/f for false): t
 New Parameter : #16<a PARAMETER>
 New Parameter List: NIL
 The attribute is Limit
Is this an input parameter? ~%: (T/t for true, F/f for false): t
Is this an output parameter? ~%: (T/t for true, F/f for false): f
 New Parameter : #16<a PARAMETER>
 New Parameter List: NIL
Give the pre condition: : Value < Limit
Give the post condition: : Value > 0
New Operation: #17<a GOMT-OP>
Are you done? : (T/t for true, F/f for false): T
.> (mcn 17)
##r GOMT
  COUNTBACKWARDS abstract?: RE::*UNDEFINED* class op?:
    RE::*UNDEFINED*
```

```
.> (pup)
#17<a gomt-op>
   class:  GOMT-OP
   has-name:  COUNTBACKWARDS
   has-parameters: []
.> (dribble)
```

Appendix E. Z Specifications Used in Testing

E.1 JetEngine

```
\begin{zed}
[ MODEL_TYPE ]
\end{zed}\\

\begin{schema}{ JetEngine }%ObjectTheory
  manufacturer : \seq CHAR\\
  model_num : MODEL_TYPE\\
  engine_weight : \Re\\
  maximum_fuel_flow_rate: \Re\\
  thrust_factor : \Re\\
  current_fuel_flow_rate: \Re\\
  current_thrust: \Re
\where
  engine_weight > 0 \\
  maximum_fuel_flow_rate > 0\\
  thrust_factor > 0\\
  current_thrust \geq 0\\
  current_fuel_flow_rate \geq 0\\
  current_fuel_flow_rate \leq maximum_fuel_flow_rate\\
  current_thrust = thrust_factor * current_fuel_flow_rate
\end{schema}\\

\begin{schema}{ Off }%StateTheory
  JetEngine
\where
  current_fuel_flow_rate = 0
\end{schema}\\

\begin{schema}{ Running }%StateTheory
  JetEngine
\where
  current_fuel_flow_rate > 0
\end{schema}\\

\begin{schema}{ ChangeFuelFlow }%EventTheory
  flow_rate: \Re
\where
  flow_rate > 0
```

```
\end{schema}\\

\begin{schema}{ ChangeThrust }%EventTheory
  thrust: \Re
\where
  thrust > 0.0
\end{schema}\\

\begin{schema}{ StartUse }%EventTheory
  flow_rate: \Re
\where
  flow_rate > 0
\end{schema}\\

\begin{schema}{ StopUse }%EventTheory
\where
  True
\end{schema}\\

\label{ JetEngine }%TableTheory
\begin{tabular}{|l|l|l||l|l|l|}
\hline
Current & Event & Guard & Next & Action & Send\\
\hline\hline
Off & ChangeFuelFlow & $ flow_rate > 0 $ & Running &
  set_rate & ChangeThrust;StartUse\\
\hline
Running & ChangeFuelFlow & $ flow_rate > 0 $ & Running &
  set_rate & ChangeThrust;StopUse;StartUse\\
Running & ChangeFuelFlow & $ flow_rate = 0 $ & Off &
  set_rate & ChangeThrust;StopUse\\
\hline\hline
\end{tabular}\\

\begin{schema}{ SetRate }%FunctionalTheory
  \Delta JetEngine\\
  flow_rate?: \Re
\where
  current_fuel_flow_rate' = flow_rate?\\
  current_thrust' = thrust_factor' * current_fuel_flow_rate'
\end{schema}
```

E.2 JetPropulsionSys

```
\begin{schema}{ JetPropulsionSys }%ObjectTheory
  fueltank : FuelTank\\
  throttle : Throttle\\
  jetengine : JetEngine\\
  prop_weight: \Re\\
  prop_fuel: \Re
\where
  prop_weight = fueltank.fuel_tank_weight + jetengine.engine_weight\\
  prop_fuel = fueltank.fuel_level\\
  (fueltank.fuel_level = 0 \implies throttle.maximum_flow_rate = 0)\\
  (fueltank.fuel_level > 0 \implies
      throttle.maximum_flow_rate = jetengine.maximum_fuel_flow_rate) \\
  fueltank.output_flow_rate = throttle.actual_flow_rate\\
  throttle.actual_flow_rate = jetengine.input_flow_rate
\end{schema}\\

\begin{schema}{ Fueled }%StateTheory
  JetPropulsionSys
\where
  fueltank.fuel_level > 0\\
  jetengine.current_fuel_flow_rate = 0
\end{schema}\\

\begin{schema}{ Operating }%StateTheory
  JetPropulsionSys
\where
  fueltank.fuel_level > 0\\
  jetengine.current_fuel_flow_rate > 0
\end{schema}\\

\begin{schema}{ NoFuel }%StateTheory
  JetPropulsionSys
\where
  fueltank.fuel_level = 0
\end{schema}\\

\begin{schema}{ StartEngines }%EventTheory
  throt_pos: \Re
\where
  throt_pos \geq 0.0\\
  throt_pos \leq 1.0
\end{schema}\\
\begin{schema}{ ChangeThrottle }%EventTheory
  throt_pos: \Re
\where
  throt_pos \geq 0.0\\
  throt_pos \leq 1.0
\end{schema}\\

\begin{schema}{ OutOfFuel }%EventTheory
\where
  True
\end{schema}\\

\begin{schema}{ ChangeSetting }%EventTheory
  new_setting: \Re
\where
  0.0 \leq new_setting \leq 1.0
\end{schema}\\

\begin{schema}{ TankEmpty }%EventTheory
\where
  True
\end{schema}\\

\label{ JetPropulsionSys }%TableTheory
\begin{tabular}{|l|l|l||l|l|l|}
\hline
Current & Event & Guard & Next & Action & Send\\
\hline\hline
% Replicate and fill in next line for each transition.
init & startup & & Fueled & jet_prop_init & \\
\hline
Fueled & StartEngines & $ throt_pos > 0.0 $ & Operating & & ChangeSetting\\
\hline
Operating & ChangeThrottle & $ throt_pos > 0.0 $ & Operating & &
    ChangeSetting\\
Operating & ChangeThrottle & $ throt_pos = 0.0 $ & Fueled & & ChangeSetting\\
Operating & OutOfFuel & & NoFuel & & TankEmpty\\
\hline
NoFuel & & & NoFuel & & \\
\hline\hline
\end{tabular}\\

\begin{schema}{ LoadFuel }%FunctionalTheory
  \Delta JetPropulsionSys\\
  fuel_load?: \Re
\where
  fueltank'.fuel_level = fuel_load?
\end{schema}
```

E.3 Throttle

```
\begin{schema}{ Throttle }%ObjectTheory
  position_index: \Re\\
  maximum_flow_rate: \Re\\
  actual_flow_rate: \Re
\where
  position_index \geq 0.0\\
  position_index \leq 1.0\\
  actual_flow_rate = position_index * maximum_flow_rate
\end{schema}\\

\begin{schema}{ InitThrottle }%FunctionalTheory
  \Delta Throttle\\
\where
  position_index' = 0.0
\end{schema}\\

\begin{schema}{ Normal }%StateTheory
  Throttle
\where
  True
\end{schema}\\

\begin{schema}{ ChangeSetting }%EventTheory
  new_setting: \Re
\where
  0.0 \leq new_setting \leq 1.0
\end{schema}\\

\begin{schema}{ ChangeFuelFlow }%EventTheory
  flow_rate: \Re
\where
  flow_rate > 0
\end{schema}\\
\label{ Throttle }%TableTheory
\begin{tabular}{|l|l|l||l|l|l|}
\hline
Current & Event & Guard & Next & Action & Send\\
\hline\hline
Normal & ChangeSetting & & Normal & update_position_index &
ChangeFuelFlow\\
\hline\hline
\end{tabular}
```

E.4 FuelTank

```
\begin{zed}
[ SIMTIME ]
\end{zed}\\

\begin{schema}{ FuelTank }%ObjectTheory
  tank_sim_time: SIMTIME\\
  input_flow_rate: \Re\\
  output_flow_rate: \Re\\
  fuel_level: \Re\\
  capacity: \Re\\
  tank_weight: \Re\\
  fuel_density: \Re\\
  fuel_tank_weight: \Re
\where
  fuel_level \leq capacity\\
  fuel_tank_weight = tank_weight + fuel_density * fuel_level
\end{schema}\\

\begin{schema}{ InitFuelTank }%FunctionalTheory
  \Delta FuelTank\\
\where
  tank_sim_time' = 0\\
  input_flow_rate' = 0\\
  output_flow_rate' = 0\\
  fuel_level' = 0\\
  capacity' = 0\\
  tank_weight' = 0\\
  fuel_density' = 0\\
  fuel_tank_weight' = 0
\end{schema}\\

\begin{schema}{ DetermineInterval }%FunctionalTheory
  \Xi FuelTank\\
  \Xi SimClock\\
  interval!: SIMTIME
\where
  interval! = sim_time - tank_sim_time
\end{schema}\\

\begin{schema}{ PredictTankFullTime }%FunctionalTheory
  \Xi FuelTank\\
  overflow_event_time!: SIMTIME
\where
  overflow_event_time! =
tank_sim_time + capacity - fuel_level \div input_flow_rate
\end{schema}\\

\begin{schema}{ CalculateNetFlow }%FunctionalTheory
  \Xi FuelTank\\
  net_flow_rate!: \Re
\where
  net_flow_rate! = input_flow_rate - output_flow_rate
\end{schema}\\

\begin{schema}{ CalculateNewLevel }%FunctionalTheory
  \Delta FuelTank\\
  \Xi SimClock\\
  net_flow_rate?: \Re\\
  interval?: SIMTIME
\where
  fuel_level' = fuel_level + interval? * net_flow_rate?\\
  tank_sim_time' = sim_time
\end{schema}\\

\begin{schema}{ PredictTankEmptyTime }%FunctionalTheory
  \Xi FuelTank\\
  tank_empty_event_time!: SIMTIME
\where
  tank_empty_event_time! =
tank_sim_time + fuel_level div output_flow_rate
\end{schema}\\

\begin{schema}{ DetermineFuelWeight }%FunctionalTheory
```

```
  \Xi FuelTank\\
  fuel_weight!: \Re
\where
  fuel_weight! = fuel_level * fuel_density
\end{schema}\\

\begin{schema}{ CalcTotalWeight }%FunctionalTheory
  \Xi FuelTank\\
  fuel_weight?: \Re\\
  fuel_tank_weight!: \Re
\where
  fuel_tank_weight! = fuel_weight? + tank_weight
\end{schema}\\

\begin{schema}{ SetInflow }%FunctionalTheory
  \Delta FuelTank\\
  \Xi SimClock\\
  flow_rate?: \Re
\where
  input_flow_rate' = flow_rate?\\
  tank_sim_time' = sim_time
\end{schema}\\

\begin{schema}{ SetOutflow }%FunctionalTheory
  \Delta FuelTank\\
  \Xi SimClock\\
  flow_rate?: \Re
\where
  output_flow_rate' = flow_rate?\\
  tank_sim_time' = sim_time
\end{schema}\\

\begin{schema}{ Empty }%StateTheory
  FuelTank
\where
  fuel_level = 0\\
  input_flow_rate = 0\\
  output_flow_rate = 0
\end{schema}\\

\begin{schema}{ PartiallyFilled }%StateTheory
  FuelTank
\where
  fuel_level > 0\\
  fuel_level < capacity\\
  input_flow_rate = 0\\
  output_flow_rate = 0
\end{schema}\\

\begin{schema}{ Full }%StateTheory
  FuelTank
\where
  fuel_level = capacity\\
  input_flow_rate = 0\\
  output_flow_rate = 0
\end{schema}\\

\begin{schema}{ Filling }%StateTheory
  FuelTank
\where
  fuel_level \geq 0\\
  fuel_level \leq capacity\\
  input_flow_rate > 0\\
  output_flow_rate = 0
\end{schema}\\

\begin{schema}{ Using }%StateTheory
  FuelTank
\where
  fuel_level \geq 0\\
  fuel_level \leq capacity\\
  input_flow_rate = 0\\
  output_flow_rate > 0
\end{schema}\\

\begin{schema}{ FillAndUse }%StateTheory
  FuelTank
\where
  fuel_level \geq 0\\
  fuel_level \leq capacity\\
  input_flow_rate > 0\\
  output_flow_rate > 0
\end{schema}\\

\begin{schema}{ StartFill }%EventTheory
  flow_rate: \Re
\where
  flow_rate > 0
\end{schema}\\
\begin{schema}{ StopFill }%EventTheory
\where
  True
\end{schema}\\

\begin{schema}{ StartUse }%EventTheory
  flow_rate: \Re
\where
  flow_rate > 0
\end{schema}\\

\begin{schema}{ StopUse }%EventTheory
\where
  True
\end{schema}\\

\begin{schema}{ TankFull }%EventTheory
\where
  True
\end{schema}\\

\begin{schema}{ TankEmpty }%EventTheory
\where
  True
\end{schema}\\

\begin{schema}{ Schedule }%EventTheory
  sched_type: EVENT_TYPE\\
  sched_time: SIMTIME
\where
  True
\end{schema}\\

\begin{schema}{ Cancel }%EventTheory
  sched_type: EVENT_TYPE\\
  sched_time: SIMTIME
\where
  True
\end{schema}\\

\begin{schema}{ Overflow }%EventTheory
\where
  True
\end{schema}\\
\begin{schema}{ ChangeFuelFlow }%EventTheory
  flow_rate: \Re
\where
  flow_rate > 0
\end{schema}\\

\label{ FuelTank }%TableTheory
\begin{tabular}{|l|l|l||l|l|l|}
\hline
Current & Event & Guard & Next & Action & Send\\
\hline\hline
Empty & StartFill & & Filling & set_inflow & Schedule\\
\hline
Filling & StopFill & $ fuel_level = capacity $ &  Full &
set_update_level & Cancel\\
Filling & StopFill & $ fuel_level < capacity $ &  PartiallyFilled &
  set_update_level & Cancel\\
Filling & TankFull & & Full & update_level & Overflow \\
Filling & StartUse & & FillAndUse & set_outflow_level & Cancel\\
\hline
Full & StartFill & & Full & & Overflow\\
Full & StartUse & & Using & set_outflow & Schedule\\
\hline
Using & TankEmpty & & Empty & update_level & ChangeFuelFlow\\
Using & StopUse & & PartiallyFilled & set_outflow_level & Cancel\\
Using & StartFill & & FillAndUse & set_inflow_level & Cancel\\
\hline
PartiallyFilled & StartFill & & Filling & set_inflow & Schedule\\
PartiallyFilled & StartUse & & Using & set_outflow & Schedule\\
\hline
FillAndUse & StopUse & & Filling & set_outflow_level & Schedule\\
FillAndUse & StopFill & & Using  & set_inflow_level & Schedule\\
\hline\hline
\end{tabular}
```

E.5 Counter

```
    \begin{schema}{ Counter }%ObjectTheory
      Value  : \nat\\
      Limit  : \nat
    \where
      Value \leq Limit
```

```
\end{schema}\\
\begin{schema}{ InitCounter }%StateTheory
  Counter
\where
  Value = 0\\
  Limit = 100
\end{schema}\\
\begin{schema}{ Increment }%FunctionalTheory
  \Delta Counter\\
\where
  Value < Limit
\also
  Value' = Value + 1
\end{schema}\\
\begin{schema}{ Add }%FunctionalTheory
  \Delta Counter\\
  AddValue?  :  \nat\\
  NewValue!  :  \nat
\where
  Value + AddValue? \leq Limit
\also
  Value' = Value + AddValue?\\
  NewValue! = Value'
\end{schema}\\
\begin{schema}{ Subtract }%FunctionalTheory
  \Delta Counter\\
  DecValue?  :  \nat\\
  NewValue!  :  \nat
\where
  Value - DecValue? \geq 0
\also
  Value' = Value - DecValue?\\
  NewValue! = Value'
\end{schema}
```

Vita

Captain Timothy Karagias [redacted]. He enlisted in the United States Air Force in 1983. His first assignment was at Barksdale AFB Louisiana where he worked as a Computer Operator. After cross training to computer programming, he was sent to Hickam AFB in Hawaii where he served as an Intelligence System Programmer. While in Hawaii, he received a Bachelor of Science in Computer Science from Hawaii Pacific College. After receiving a commission in 1990, he became a Software Test Engineer at Scott AFB in Illinois. From there, he went to Offutt AFB in Nebraska where he developed software for the war planning system. He entered the School of Engineering, Air Force Institute of Technology in May, 1995.

[redacted]

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1996	3. REPORT TYPE AND DATES COVERED final

4. TITLE AND SUBTITLE Elicitation of Formal Software Specifications from an Object-Oriented Domain Model	5. FUNDING NUMBERS
6. AUTHOR(S) Timothy Karagias	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/96D-14
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Mark Gerken Rome Laboratory 525 Brooks Rd Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The ability to provide automated support for the generation of formal software specifications would lead to decreased software development time. By *eliciting* the needed information from a software developer and *harvesting* the proper parts of a domain model, a software specifications document could be created. This research establishes the feasibility of producing customized software specifications based on an object-oriented domain model. The research was conducted in three phases. The first phase was to define the requirements for the Elicitor Harvester. Those requirements were balanced between the capabilities of the existing Knowledge Based Software Engineering (KBSE) software used at AFIT and the needs of the Elicitor Harvester system. The second phase consisted of creating a design capable of meeting those requirements. The design was open enough to use the existing software and flexible enough to evolve in an incremental manner. The final phase involved the implementation and testing of a feasibility demonstration of the Elicitor Harvester system. Specifications were successfully generated from two significantly different domain models.

14. SUBJECT TERMS Domain Modeling, Elicitor Harvester	15. NUMBER OF PAGES 105
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL